SENAI CIMATEC

PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL E TECNOLOGIA INDUSTRIAL

Mestrado em Modelagem Computacional e Tecnologia Industrial

Dissertação de mestrado

UM AMBIENTE COMPUTACIONAL TOLERANTE A FALHAS PARA APLICAÇÕES PARALELAS

Apresentada por: Oberdan Rocha Pinheiro
Orientador: Dr. Josemar Rodrigues de Souza

Fevereiro/2013

Oberdan Rocha Pinheiro

UM AMBIENTE COMPUTACIONAL TOLERANTE A FALHAS PARA APLICAÇÕES PARALELAS

Dissertação de mestrado apresentada ao Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial, Curso de Mestrado em Modelagem Computacional e Tecnologia Industrial do SENAI CIMATEC, como requisito parcial para a obtenção do título de **Mestre em Modelagem Computacional e Tecnologia Industrial**.

Área de conhecimento: Interdisciplinar

Orientador: Dr. Josemar Rodrigues de Souza
SENAI CIMATEC

Salvador
SENAI CIMATEC
2013

Nota sobre o estilo do PPGMCTI

Esta dissertação de mestrado foi elaborada considerando as normas de estilo (i.e. estéticas e estruturais) propostas aprovadas pelo colegiado do Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial e estão disponíveis em formato eletrônico (*download* na Página Web http://ead.fieb.org.br/portal_faculdades/dissertacoes-e-teses-mcti.html ou solicitação via e-mail à secretaria do programa) e em formato impresso somente para consulta.

Ressalta-se que o formato proposto considera diversos itens das normas da Associação Brasileira de Normas Técnicas (ABNT), entretanto opta-se, em alguns aspectos, seguir um estilo próprio elaborado e amadurecido pelos professores do programa de pós-graduação supracitado.

SENAI CIMATEC

Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial

Mestrado em Modelagem Computacional e Tecnologia Industrial

A Banca Examinadora, constituída pelos professores abaixo listados, leram e recomendam a aprovação [com distinção] da Dissertação de mestrado, intitulada "UM AMBIENTE COMPUTACIONAL TOLERANTE A FALHAS PARA APLICAÇÕES PARALELAS", apresentada no dia 04 de Fevereiro de 2013, como requisito parcial para a obtenção do título de **Mestre em Modelagem Computacional e Tecnologia Industrial**.

Orientador:

Prof. Dr. Josemar Rodrigues de Souza
SENAI CIMATEC

Membro externo da Banca:

Prof. Dr. Ângelo Amâncio Duarte
Universidade Estadual de Feira de Santana

Membro externo da Banca:

Prof. Dr. Marcos Ennes Barreto
Universidade Federal da Bahia

Membro interno da Banca:

Prof. Dr. Hernane Borges de Barros Pereira
SENAI CIMATEC

Dedico este trabalho aos meus pais, Pinheiro e Maria Lucia, aos meus irmãos Reinaldo e Alin Tatiana, aos meus filhos, Pedro e Olívia, e à minha esposa, Glacir.

Agradecimentos

A Josemar Rodrigues de Souza, pela orientação e paciência que dedicou a este trabalho, pelas sugestões valiosas, conduzindo, de forma coerente, a investigação e a elaboração desta dissertação de mestrado.

Aos professores do Programa de Pós-Graduação Stricto Sensu da Faculdade de Tecnologia SENAI CIMATEC, pelo incentivo e questionamentos, que foram fundamentais no processo da pesquisa e desenvolvimento desse trabalho.

À minha esposa e aos meus filhos, que, de forma grandiosa, me incentivaram a continuar, não importando quais fossem os problemas nos momentos difíceis dessa caminhada.

A todos os meus parentes e familiares, pelo apoio, incentivo e compreensão, ante a minha ausência em muitos momentos de confraternização.

Aos meus alunos do CST em Mecatrônica Industrial do SENAI CIMATEC, que souberam compreender a minha ausência às aulas, quando, em muitos momentos, estive em reunião com o meu orientador.

Salvador, Brasil
04 de Fevereiro/2013

Oberdan Rocha Pinheiro

Resumo

O desempenho computacional disponibilizado pelos sistemas paralelos resulta da capacidade de dividir o trabalho em partes menores e encaminhar cada uma delas para ser processada paralelamente em diferentes nós de um sistema distribuído. A falha em uma das partes paralelizadas pode levar a computação a um estado de operação inadequado, comprometendo o resultado final da computação paralela distribuída. Um sistema distribuído está sujeito a falhas nos seus componentes de comunicação, seus processadores, em suas aplicações entre outros componentes que formam o sistema. Desta maneira, as aplicações paralelas, ao utilizarem os recursos disponibilizados pelos sistemas distribuídos, têm suas partes executadas em paralelo, em diferentes nós desse sistema. Em razão de cada um desses recursos ser um possível ponto de falha, as aplicações paralelas acabam se tornando mais susceptíveis à ocorrência de falhas. Quando as aplicações paralelas são interrompidas durante a ocorrência de falhas, todo o processamento realizado e o tempo gasto para tal são desperdiçados, pois as aplicações devem ser reinicializadas. Dessa forma, o desenvolvimento de técnicas de tolerância a falhas torna-se fundamental, para garantir o término das aplicações paralelas. Este trabalho apresenta um ambiente computacional tolerante a falhas para aplicações paralelas que utilizam o padrão Open MPI, para minimizar o desperdício de tempo e processamento já realizados pelos processos da aplicação paralela, até o momento do surgimento da falha. O ambiente utiliza mecanismo de *checkpoint/restart* do padrão Open MPI para armazenar e recuperar os estados dos processos paralelos e a técnica de *heartbeat* para verificar a continuidade de execução destes mesmos processos.

Palavras-chave: Tolerância a Falhas, Protocolos de *Checkpoint*, MPI.

Abstract

The computational performance provided by parallel systems results from the ability to divide the job into smaller pieces and send each of it to be processed in parallel on different nodes of a distributed system. The failure of any part in a parallelized computation can lead to an improper state of operation, compromising the final result of the parallel distributed computing. In a distributed system failures can occur in communication components, their processors and in its applications among other components that form the system. Thus, for parallel applications to utilize the resources provided by distributed systems, its parts are performed in parallel on different nodes of the system. Because each of these resources being a potential failure point, parallel applications eventually become more susceptible to failures. When parallel applications are interrupted during the occurrence of faults, all the processing performed and time spent are wasted because applications must be restarted. By the way, the development of techniques for fault-tolerance becomes critical to ensure the completion of parallel applications. This paper presents a computational environment for fault-tolerant parallel applications that use the standard Open MPI to minimize wasted time and processing already performed by its processes until the time the failure occur. The environment uses mechanism checkpoint/restart in Open MPI standard for storing and retrieving the states of parallel processes and technique heartbeat to check the continuity of implementing these same processes.

Keywords: Fault tolerance, Checkpoint protocols, MPI.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Siglas

ACTF	Ambiente Computacional Tolerante a Falhas
AP	Aplicação Paralela
BLCR	Berkeley Labs Checkpoint/Restart
Cimatec	Centro Integrado de Manufatura e Tecnologia
CRCP	Checkpoint/Restart Coordination Protocol
CRS	Checkpoint/Restart Service
CST	Curso Superior de Tecnologia
DAO	Data Access Object
DF	Detector de Falhas
FTDR	Fault Tolerant Data Replication
GD	Gestor de Dados
GP	Gestor de Processos
GUI	Graphical User Interface
HPC	High Performance Computing
MCA	Modular Component Architecture
MPI	Message Passing Interface
NFS	Network File System
OMPI	Open MPI layer
OPAL	Open Portable Access Layer
ORTE	Open RunTime Environment
PPGMCTI ..	Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial
SATF	Sistema de Arquivos Tolerante a Falhas
Senai	Serviço Nacional de Aprendizagem Industrial
SnapC	Snapshot Coordinator
SPMD	Single Program Multiple Data

Capítulo Um

Introdução

Pesquisadores usam computação de alto desempenho, para resolver problemas científicos complexos. Os sistemas de computação de alto desempenho fornecem recursos computacionais para distribuir a aplicação em um ambiente de computação paralela colaborativa. Quanto maior o número de componentes envolvidos nesse ambiente, menos confiáveis tornam-se esses sistemas (HURSEY, 2010). Caso a aplicação não esteja preparada para tratar as falhas ocasionadas pela falta de algum componente dos sistemas de computação de alto desempenho, ela corre o risco de perder todo o processamento realizado até o momento da falha, que pode ter levado horas ou dias para gerar a computação desejada.

O tempo de execução das aplicações paralelas tornou-se importante, porque os problemas do mundo real exigem resultados no menor tempo possível, e esse fato demanda alto poder de computação. Falhas podem custar muito em aplicações, como simulações de sistemas aeroespaciais, meteorológicos, protótipos de *hardware* e aplicações de processamento de imagem digital, em que o processamento de imagens requer grande poder de computação, e a falha de processamento, nessas aplicações, pode resultar em perdas econômicas.

Muitas técnicas foram desenvolvidas para o aumento da confiabilidade e alta disponibilidade dos sistemas paralelos e distribuídos. As técnicas mais utilizadas incluem as de comunicações em grupo, replicação de dados e recuperação retroativa. Cada uma delas envolve diferentes relações de compromisso, com o objetivo de tolerar falhas de maneiras diferentes.

1.1 Definição do problema

O custo computacional para solucionar os grandes problemas da área científica, fazendo uso de aplicações sequenciais, pode ser proibitivo, dado que essas aplicações demandam alto poder de processamento. Sendo assim, torna-se necessária a introdução das aplicações paralelas distribuídas, com o objetivo de solucionar estes problemas. Desta forma, as aplicações paralelas distribuídas, quando colocadas em execução, utilizam as funcionalidades disponibilizadas pelo ambiente distribuído, tais como a distribuição das tarefas a serem executadas em vários processadores e a utilização do sistema de comunicação e também o acesso ao sistema de arquivos. Justamente por utilizar estes recursos distribuídos e, muitas vezes, não preparados para tratar as falhas, as aplicações paralelas distribuídas estão susceptíveis a serem interrompidas, em decorrência de uma falha em

algum destes componentes.

A comunidade de computação de alto desempenho não tem dado muita atenção para o problema da tolerância a falhas em plataformas paralelas devido ao fato de que as máquinas paralelas tradicionais não são afetadas frequentemente pelas falhas, sejam elas de *hardware* ou de *software* (BRONEVETSKY et al., 2003).

Entre as técnicas para prover tolerância a falhas do MPI, as que se sobressaem em número de implementações são as de recuperação retroativa (ELNOZAHY et al., 2002). Nas técnicas de recuperação retroativa, a computação da aplicação retrocede para um ponto que representa o estado global consistente da aplicação, sempre que uma falha ocorre. No *checkpoint* coordenado, o estado de todos os processos é guardado periodicamente em mídia confiável, para posterior restauração simultânea, em caso de falha. O *checkpoint* coordenado funciona de maneira que o conjunto de estados armazenados forma um estado global parcial consistente, que pode ser restaurado sem risco de inconsistências na troca de mensagens entre os processos da aplicação.

Outra técnica para se obter a tolerância a falhas no nível de *software*, é a utilização da redundância de código, ou seja, ter o mesmo código executado nos nós dos processos da aplicação paralela (SOUZA, 2006).

As falhas impactam de maneira crítica o funcionamento dos sistemas, essas falhas podem ser classificadas nas classes conhecidas por *crash failure*, *fail stop* e *bizantino* (CRISTIAN, 1991). É importante a criação de mecanismos que permitam, às aplicações paralelas distribuídas que utilizam MPI, poderem terminar a computação corretamente, mesmo na presença de falhas, possibilitando ao usuário programador, desenvolver o seu programa em MPI sem a preocupação com o término da computação na presença de falhas.

1.2 Objetivo

O objetivo deste trabalho é construir um ambiente computacional tolerante a falhas, para aplicações paralelas que utilizam o padrão Open MPI, fornecendo mecanismos para minimizar o impacto sobre o desempenho das aplicações paralelas distribuídas, quando as falhas ocorrerem ao longo da execução da aplicação paralela. Os objetivos específicos são:

- Desenvolver um subsistema de detecção de falhas da classe *fail stop*, utilizando a estratégia de monitoramento descentralizada e permitindo que os nós participantes do *cluster* de computadores possam ser monitorados.

- Desenvolver um subsistema que utilize a técnica de *checkpoint/restart* coordenado para armazenar e recuperar os estados dos processos.
- Avaliar experimentalmente o ambiente proposto.

1.3 Importância da pesquisa

MPI (*Message Passing Interface*) é um padrão de interface de passagem de mensagens para comunicação de dados em computação paralela distribuída e não promove mecanismo para gerenciar falhas nos sistemas de comunicação, processadores ou falhas nos nós participantes da computação, de maneira automática para o usuário da aplicação, tornando necessário que o usuário programador obtenha conhecimentos específicos da área de tolerância a falhas, para tornar a execução da sua aplicação segura.

Os processos MPI precisam estar associados a um comunicador, para realizar a troca de mensagens, e não podem se encerrar isoladamente. O MPI implementa um mecanismo de sincronismo, através da função **MPI_Finalize()**. Esta função deve ser executada, necessariamente, ao término de cada processo MPI. Caso o processo se encerre sem executar essa função, o MPI interpretará tal situação como um erro, e toda a aplicação, ou seja, todos os processos MPI serão finalizados antes do término da computação.

Caso uma falha ocorra, todo o processamento realizado até aquele momento e o tempo gasto para tal serão desperdiçados, exigindo que as aplicações sejam reiniciadas. Assim, torna-se importante a pesquisa por mecanismos que venham a tratar tais problemas, com o objetivo de minimizar estes desperdícios de tempo e de processamento realizado.

A importância e a principal contribuição desta pesquisa consiste na possibilidade de viabilizar que aplicações paralelas distribuídas, implementadas no padrão Open MPI, executadas em um *cluster* de computadores, possam contar com mecanismos que, mesmo com o surgimento de falhas durante a execução dessas aplicações, preservem o processamento e o tempo gasto pelas mesmas, reativando-as automaticamente sem a necessidade de nenhuma intervenção do usuário programador em uma nova configuração do *cluster*.

1.4 Organização da Dissertação de mestrado

Este documento apresenta 6 capítulos e está estruturado da seguinte forma:

- **Capítulo 1 - Introdução**: Contextualiza o âmbito no qual a pesquisa proposta está

inserida. Apresenta a definição do problema, objetivos e justificativas da pesquisa e a estrutura desta dissertação de mestrado;

- **Capítulo 2 - Tolerância a Falhas**: Neste capítulo são apresentados os conceitos básicos de tolerância a falhas, assim como as principais estratégias de tolerância, através de mecanismos de *checkpoint* e protocolos de *log*;
- **Capítulo 3 - Estado da Arte**: Neste capítulo, apresentam-se algumas implementações, baseadas no padrão Open MPI, que fornecem serviços de tolerância a falhas e discute-se o estado da arte em tolerância a falhas para aplicações paralelas distribuídas;
- **Capítulo 4 - Projeto e implementação do ACTF**: Neste capítulo, é apresentado o projeto do ambiente computacional tolerante a falhas para aplicações paralelas distribuídas, que utilizam o padrão Open MPI, onde se analisa a modelagem e implementação do ambiente;
- **Capítulo 5 - Avaliação e Resultados Experimentais**: Neste capítulo, são apresentados os resultados experimentais do mecanismo de tolerância a falhas proposto;
- **Capítulo 6 - Considerações Finais**: Neste capítulo, apresentam-se as conclusões, contribuições da pesquisa e algumas sugestões de atividades de pesquisa a serem desenvolvidas no futuro.

Capítulo Dois

Tolerância a falhas

2.1 Introdução

De acordo com Weber e Weber (1989), quando o sistema exige alta confiabilidade e alta disponibilidade, técnicas de tolerância a falhas devem ser utilizadas para garantir o funcionamento correto do sistema, mesmo na ocorrência de falhas, exigindo componentes adicionais ou algoritmos especiais. Tolerância a falhas tem como objetivo permitir que um sistema comporte-se como especificado durante a ocorrência de falhas, visando minimizar o aparecimento das mesmas ou, então, tratá-las quando ocorrerem.

2.1.1 Falha, erro e defeito

Na construção do tema "tolerância a falhas", existem três termos fundamentais: falha, erro e defeito. Recursivamente, falhas são as causas dos erros, que, por sua vez, são as causas de defeitos (PRADHAN, 1996).

Uma falha está relacionada a um termo físico ou a um estado faltoso de um componente de *hardware*, como memória, por exemplo. Já erros são relacionados no universo dos dados, como uma interpretação errônea de dados, proveniente, em geral, de uma falha ocorrida anteriormente em um componente físico. Por fim, defeitos estão relacionados no aspecto do usuário, um desvio de uma especificação ou, então, a impossibilidade, por parte do usuário, de realizar uma ação, devido a um erro ocorrido nos dados pré-processados. Segundo Weber e Weber (1989), nem sempre uma falha leva a um erro, e nem sempre um erro conduz a um defeito, são estágios em separado que podem ser contidos ou simplesmente passar despercebidos, em determinadas situações. Esse universo é representado na Figura 2.1, que ilustra a abrangência e áreas de atuação de cada item especificado.

De acordo com Weber e Weber (1989), uma falha é inevitável, uma vez que um componente pode envelhecer ou sofrer influências do meio externo. Por outro lado, defeitos, utilizando-se de técnicas de tolerância a falhas, são perfeitamente passíveis de serem evitados.

Segundo Pradhan (1996), uma falha é o resultado de diversos aspectos inerentes a um componente externo, ao próprio componente durante o processo de desenvolvimento do componente ou do sistema em que o mesmo está inserido. No âmbito das possibilidades

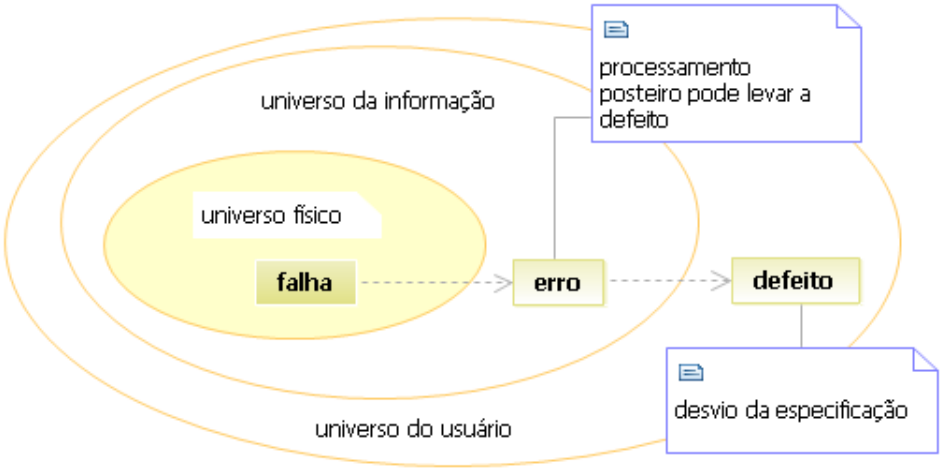

Figura 2.1: Modelo de 3 universos: falha, erro e defeito. Fonte: Autor.

de falhas, existem as falhas natas de um componente, que veio com problemas de fábrica, ou uma falha no projeto do sistema, ou até mesmo na hora da programação do sistema, bem como na incorreta especificação dos componentes que integrarão o sistema.

2.1.2 Formas de tolerância a falhas

Tomando-se como base o funcionamento e a disponibilidade de sistemas distribuídos e *clusters*, além de sua alternância entre os estados funcionando e em reparo, duas propriedades são definidas para os respectivos sistemas. Estas propriedades referem-se ao conjunto de execuções que os sistemas realizam, e são denominadas *safety* e *liveness* (GäRTNER, 1999).

A propriedade *safety* pode ser caracterizada formalmente, especificando-se quais execuções são seguras ou não, garantindo que o resultado emitido por elas esteja correto, através da execução de algum procedimento de detecção e ativação de procedimentos de correção, como mecanismos de *checksum*.

Já a propriedade *liveness* indica que o processamento parcial de um sistema estará funcionando (*live*) para um conjunto de processamento maior, se e somente se, ele puder ser estendido ou seus resultados puderem ser aproveitados pelo restante do processamento do sistema ou componente. Em outras palavras, preocupa-se em manter o sistema ou componente apto a executar, sem se preocupar com os resultados emitidos por ele, mas obrigando-o a emitir um resultado.

Em resumo, a propriedade *liveness* procura garantir que o sistema ou componente mantenha-se sempre funcionando, operacional e disponível; e a propriedade *safety* procura garantir

que o resultado da execução feita pelo componente ou sistema esteja sempre correto. A forma como o sistema se comporta em uma situação de falha e o estado que ele venha ou não garantir no decorrer da mesma determina quais propriedades (*safety* e/ou *liveness*) da tolerância a falhas e suas respectivas formas são atingidas, conforme indicado na Figura 2.2.

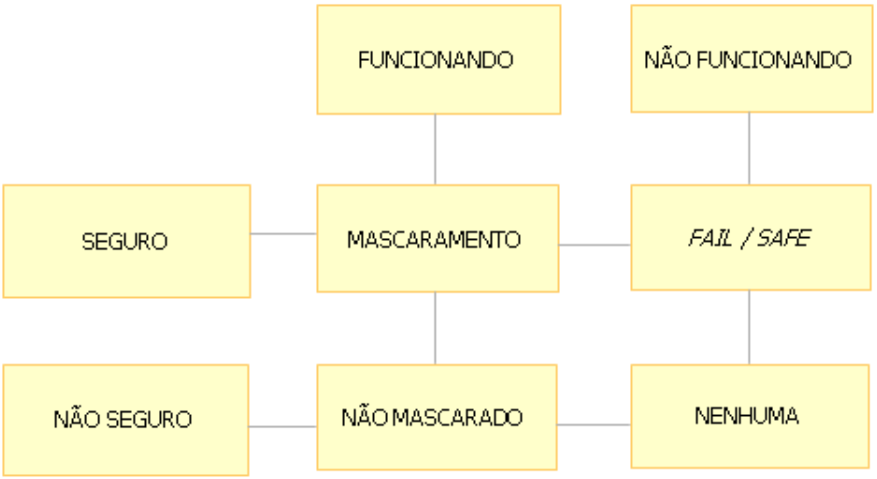

Figura 2.2: Formas de Tolerância a Falhas. Fonte: Gärtner (1999).

No mascaramento das falhas, estas não interrompem o funcionamento e não corrompem a integridade do sistema. Assim, as propriedades *safety* e *liveness* são atingidas em sua plenitude. Na forma "nenhuma" (Figura 2.2), a falha pode corromper o funcionamento e a integridade do processamento, fazendo com que as propriedades *safety* e *liveness* não sejam atingidas. O mascaramento pode ser classificado em hierárquico de falhas ou mascaramento de falhas em grupo.

No mascaramento hierárquico de falhas, um comportamento falho pode ser classificado somente com respeito a uma especificação de comportamento que se espera dele. Se um componente depende de um nível mais baixo, para fornecer um determinado serviço, então, uma falha, neste nível mais baixo, acarretará uma falha no oferecimento do serviço pelo nível mais alto. Sendo que, se esta falha for, então, mascarada, não será vista pelo nível mais alto, e o serviço em questão não será afetado (CRISTIAN, 1991).

Já no mascaramento de falhas em grupo, para se tentar garantir que um serviço continue disponível para os usuários, independentemente das falhas de seus componentes, pode-se implementar o serviço através de um grupo de componentes redundantes e fisicamente independentes. Isso garantirá que, no caso de um deles falhar, o restante continuará fornecendo o serviço. Nos casos intermediários, o não mascaramento deixa o sistema funcionando (*liveness*), mas, possivelmente, não seguro, quanto às respostas emitidas, e o *fail / safe* deixa o sistema seguro no que se refere ao resultado do processamento (*safety*), mas, possivelmente, não operacional.

Um pré-requisito para a implantação de um serviço de *software*, a ser realizado por um usuário programador ou um grupo de usuários, capaz de mascarar falhas, está na existência de múltiplos processadores (*hosts*) com acessos aos mesmos recursos físicos, utilizados pelo serviço em questão (CRISTIAN, 1991). A replicação destes recursos faz-se necessária, para tornar o serviço disponível, independentemente das falhas dos recursos individuais.

2.1.3 Técnicas de tolerância a falhas

Para alcançar tolerância a falhas em sistemas paralelos distribuídos, é importante ter mecanismos de proteção, detecção, recuperação e reconfiguração do sistema para que, em caso de falhas, o sistema possa continuar sua execução, isolando o nodo do *cluster* de computadores que falhou.

Podemos obter a tolerância a falhas em *software*, fazendo uso de alguns dos mecanismos, como redundância (AVIZIENIS et al., 2004; SOUZA, 2006), registros em *logs* (JOHNSON, 1989), pontos de verificação (PARK; YEOM, 2000; BRONEVETSKY et al., 2003).

2.1.3.1 Técnica de redundância

A tolerância a falhas é alcançada por técnicas de redundância de sistemas (AVIZIENIS et al., 2004). A redundância pode existir no nível de *hardware*, introduzindo-se um *hardware* extra; no nível de *software*, onde pode-se executar várias versões do mesmo programa; na replicação de operações e comparação de resultados e na utilização de técnicas de replicação de dados (SOUZA, 2006).

A replicação de *hardware* e *software* é um esquema de tolerância a falhas, utilizada em sistemas embarcados ou com poucos componentes, mas não é uma estratégia adequada para computadores paralelos com um grande número de componentes.

Souza (2006) propõe um modelo denominado FTDR (*Fault Tolerant Data Replication*), com o objetivo de prover tolerância a falhas em *clusters* geograficamente distribuídos, utilizando-se replicação de dados sobre uma arquitetura *Master/Worker*, conforme Figura 2.3.

O FTDR utiliza uma arquitetura onde cada *cluster* mantém uma estrutura *Master/Worker*, existe um *cluster* principal (MC) onde está o *Master* principal (MMT), encarregado de iniciar e finalizar a aplicação. Cada *cluster* do *multicluster* forma um *subcluster* com

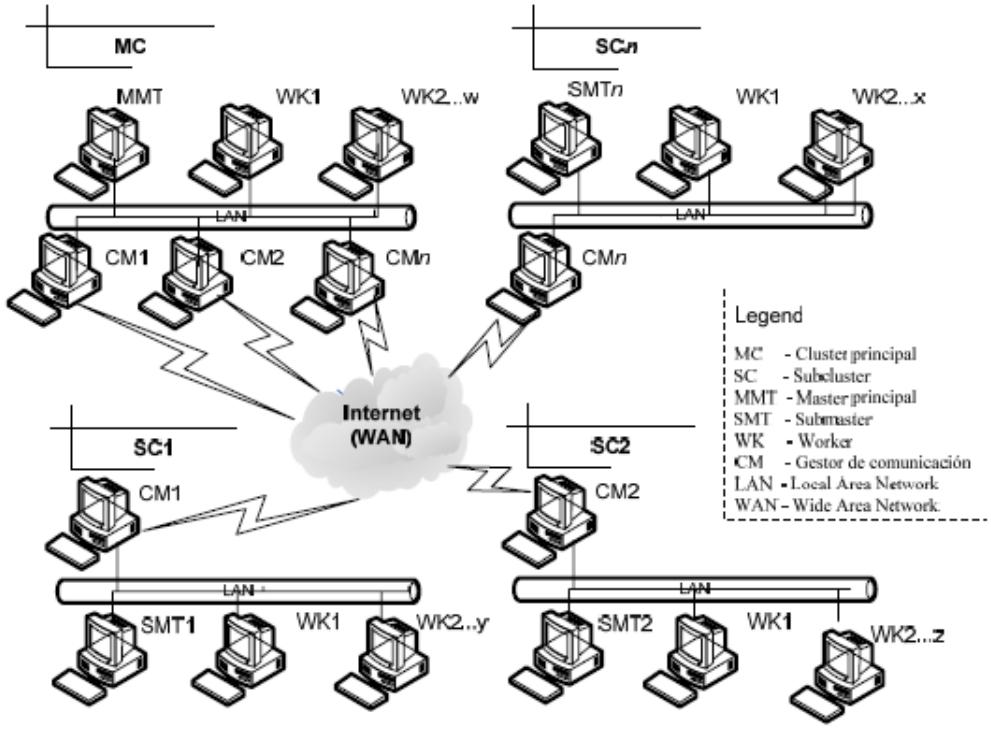

Figura 2.3: Arquitetura FTDR. Fonte: Souza (2006).

sua própria estrutura *Master/Worker*, de forma que o *subcluster* é considerado *Worker* do *Master* principal. Para a comunicação entre os *clusters* é utilizado um gestor de comunicação (CM), projetado para melhorar o rendimento e gerir a disponibilidade da interconexão entre os *cluster*, de forma que se encarrega de tratar as falhas intermitente provocadas pela internet.

Em um sistema implementado sobre o modelo de programação paralela *Master/Worker* no paradigma SPMD (*Single Program Multiple Data*), tem-se redundância implícita de código. Nesse modelo, o *Master* distribui o trabalho para os *Workers*, que processam os dados e devolvem o resultado para o *Master*, conforme indicado na Figura 2.4.

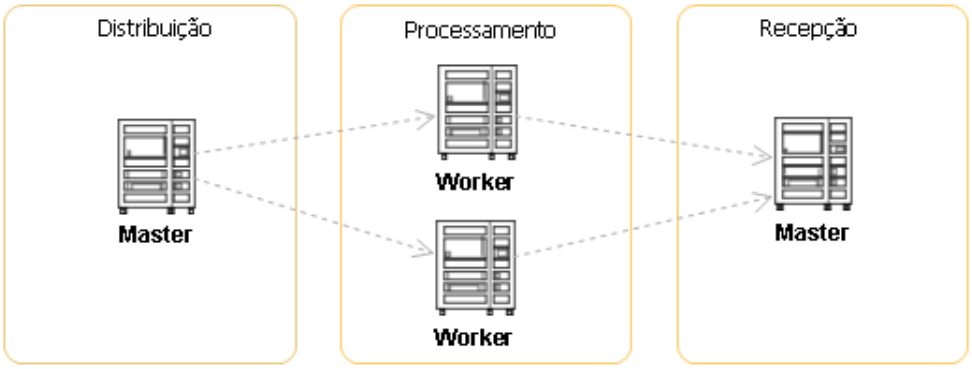

Figura 2.4: Modelo de Programação *Master/Worker*. Fonte: Autor.

2.1.3.2 Técnica de registros em Log

A técnica de registros em *log* faz uso de um sistema de armazenamento estável onde são registradas as operações realizadas pelos processos (do processamento ao envio e recebimento de mensagens); este armazenamento é feito em um arquivo, para que seja possível uma recuperação futura, no caso do surgimento de alguma falha. Nesta técnica, quando um processo apresenta uma falha, apenas ele deve ser reiniciado e executar o procedimento de *rollback*, utilizando o arquivo de *log*. Os demais processos continuam em execução como se nada tivesse ocorrido, porém podem auxiliar no processo reiniciado, manipulando (enviando ou recebendo) as mensagens utilizadas antes do surgimento da falha.

2.1.3.3 Técnica de checkpoint

A técnica de *checkpoint* permite que programas armazenem seus estados em intervalos regulares, para que possam ser reiniciados após uma falha, sem perderem o trabalho já realizado. Esta é uma técnica utilizada, para tolerar falhas transientes e para se evitar a perda total de um processamento já realizado.

Cada processo pode armazenar o seu estado individual em *checkpoints* locais, sendo uma coleção destes *checkpoints* locais denominada de *checkpoint* global (SOUZA, 2006; WANG et al., 1995; LUMPP J.E., 1998). A técnica de *checkpoint*, seja definida pelo sistema ou pelo programador, é muito utilizada em aplicações paralelas distribuídas, e as duas categorias mais utilizadas são: a coordenada e a não coordenada (BRONEVETSKY et al., 2003), ilustrada na Figura 2.5.

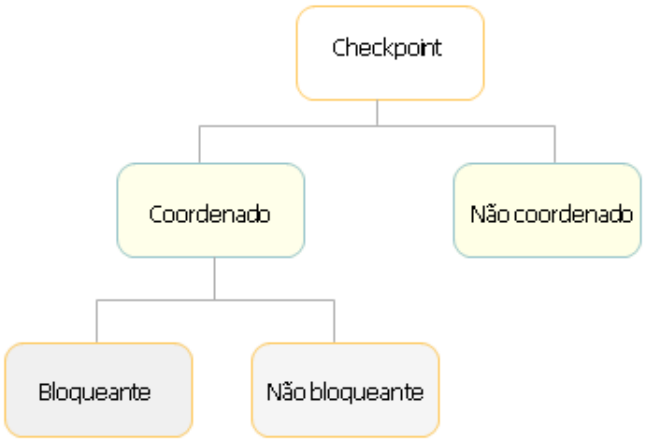

Figura 2.5: Protocolo de *checkpoint*. Fonte: Autor.

A categoria coordenada apresenta um mecanismo, para evitar estados inconsistentes após

uma recuperação, visto que todos os processos devem estar sincronizados, e as mensagens já enviadas deverão atingir seus destinos, antes que os *checkpoints* sejam realizados. Já a categoria não coordenada permite que cada processo possa decidir, de maneira independente, quando realizará seu *checkpoint*. Esta autonomia deve ser vista com cautela, pois ela pode fazer com que uma aplicação necessite reiniciar-se completamente por conta de recuperações realizadas em cascata. Estas recuperações sucessivas são chamadas de efeito dominó (ELNOZAHY et al., 2002).

A categoria de *checkpoint* coordenado não é susceptível ao efeito dominó, devido aos processos se reiniciarem sempre de um *checkpoint* global mais recente. A recuperação é simplificada, e o dispositivo de armazenamento sofre uma menor carga de trabalho do que aquela provocada pela categoria não coordenada (ELNOZAHY et al., 2002).

O *checkpoint* coordenado pode ser subdividido em bloqueante e não bloqueante. A forma bloqueante faz com que todos os processos interrompam suas execuções, antes de um *checkpoint* global ser realizado. Técnicas de bloqueio por *software* utilizam a técnica de barreira, ou seja, quando os processos atingem uma barreira global, cada um armazena seu estado local e, então, é criado um *checkpoint* global. Em seguida, os processos continuam suas execuções (BRONEVETSKY et al., 2003). Já na forma não bloqueante, os processos armazenam seus estados e suas mensagem individualmente, necessitando, para isso, de um protocolo de coordenação global. O protocolo Chandy-Lamport (CHANDY; LAMPORT, 1985) é o mais usado dos protocolos não bloqueantes (BRONEVETSKY et al., 2003).

Alguns tipos de técnicas buscam reduzir a carga produzida pelos *checkpoints* no sistema. Como exemplos destas técnicas, podem ser citados o *forked checkpointing*, o *incremental checkpointing*, o *main memory checkpointing* e o *compressed checkpointing*.

O *Forked Checkpointing* utiliza uma instrução *fork* para criar um processo filho, que se encarrega de tomar os *checkpoints*, gravando-os sempre em um mesmo arquivo, sendo que, enquanto o contexto do processo estiver sendo gravado no arquivo, o mesmo deverá estar parado ou suspenso. Isto se torna necessário em razão de se preservar o estado local do processo, ou seja, para que entre a tomada do estado local e a efetiva gravação do mesmo, não ocorra qualquer alteração. Já no *incremental checkpointing*, cria-se um arquivo de *checkpoint* principal, o qual armazena os estados, e alguns outros arquivos de *checkpoints* secundários, que armazenam as diferenças entre os estados atuais e os anteriores.

O *Main Memory Checkpointing* utiliza um *thread*, para realizar uma cópia do espaço de dados do programa no arquivo de *checkpoint*, sem interromper o processo que está sendo registrado. E o *Compressed Checkpointing* usa um utilitário de compressão de dados, para diminuir o tamanho do arquivo de *checkpoint*.

As técnicas de *checkpoint* e registro em log podem ser classificadas sob dois esquemas, conhecidos como pessimista e otimista (MAIER, 1993). O modo pessimista registra cada processamento, transmissão ou recepção de mensagens que o processo realizar sincronamente. Isto aponta para uma rápida recuperação, porém aponta, também, para uma sobrecarga no sistema, ao se registrarem praticamente todas as ações, e isso pode ser proibitivo, uma vez que pode ocasionar uma diminuição no desempenho das aplicações. O modo pessimista assim é conhecido por assumir uma alta probabilidade de ocorrência de falhas em sistemas paralelos distribuídos.

Já o modo otimista registra o processamento, as transmissões e recepções, apenas, de tempos em tempos, assincronamente. Isto causa uma pequena sobrecarga durante o processamento normal da aplicação, e permite que os processos determinem o momento em que se deve ou não registrar seus estados e mensagens. O termo otimista vem do pressuposto de que o aparecimento de falhas tem uma baixa probabilidade de ocorrer em sistemas paralelos distribuídos.

A recuperação, para ambos os modos, é feita, apenas, quando se deseja recuperar o estado armazenado no último *checkpoint*.

2.1.4 Mecanismo de monitoramento de falhas

Grande parte dos problemas relacionados aos sistemas distribuídos requer algum tipo de coordenação entre os diversos componentes (GREVE, 2005).

As principais estratégias de monitoramento utilizadas por detectores de falhas são a *push* e a *pull*. Na abordagem *push*, o processo monitorado toma um papel mais ativo, enquanto que, em uma abordagem *pull*, ele adota um papel mais passivo (CHEN; TOUEG; AGUILERA, 2002). Segundo Chen, Toueg e Aguilera (2002), detectores de falhas que utilizem o paradigma *push* possuem alguns benefícios, se comparados àqueles que utilizem uma abordagem *pull*, pois os primeiros necessitam de apenas a metade das mensagens para uma qualidade de detecção equivalente.

2.1.4.1 Estratégia Push

O mecanismo ilustrado na Figura 2.6 funciona da seguinte maneira: seja *Worker* o processo monitorado, e *Master*, o processo detector de falhas. O processo *Worker* envia periodicamente um *heartbeat* ("estou vivo") para o processo *Master*. Quando o *Master* recebe um *heartbeat*, ele confia no *Worker* por um período de tempo, porém, caso novas

mensagens não cheguem até expirar este *timeout*, o *Master* assume que ocorreu uma falha do processo *Worker*.

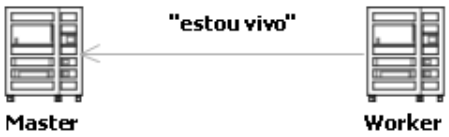

Figura 2.6: Monitoramento por *push*. Fonte: Autor.

2.1.4.2 Estratégia Pull

Esta estratégia é também baseada na troca de mensagens periódicas, para suspeitar acerca de falhas em processos, porém estas mensagens são do tipo *query-response*, também conhecidas como *ping*. O mecanismo ilustrado na Figura 2.7 funciona da seguinte maneira: seja *Worker* o processo monitorado e *Master* o processo detector de falhas. O processo *Master* envia periodicamente uma *query* ("você está vivo?") para o processo *Worker*, que deve responder ("eu estou vivo") em tempo hábil, porém, caso tal resposta não chegue em um determinado *timeout*, o *Master* assume que ocorreu uma falha do processo *Worker*.

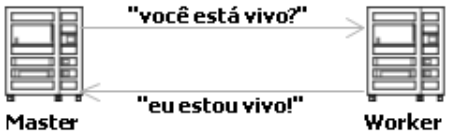

Figura 2.7: Monitoramento por *pull*. Fonte: Autor.

2.1.4.3 Qualidade do serviço de detecção de falhas

Um detector de falhas pode ser definido por duas características:

- Completude - é a garantia de que a falha em um membro do grupo é eventualmente detectada por cada outro membro não falho.
- Eficiência - significa que as falhas são detectadas rápida e precisamente.

Algumas aplicações distribuídas confiam em um único computador central ou, mesmo, em poucos computadores centrais, para a detecção de falhas ao longo da computação. Esses computadores são responsáveis por manter as informações ao longo de toda a computação

e, portanto, a eficiência na detecção de uma falha depende do tempo em que a falha é inicialmente detectada por um membro não falho.

Gupta, Chandra e Goldszmidt (2001) observou que, mesmo na falta de um servidor central, a notificação de uma falha é tipicamente comunicada a todo o grupo pelo primeiro membro a detectá-la, via broadcast. Portanto, mesmo sendo importante alcançar completude, uma eficiente detecção de falhas é mais comumente relacionada com o tempo da primeira detecção da falha.

Chandra e Toueg (1996) analisaram as propriedades dos detectores de falhas, demonstrando porque é impossível, para um algoritmo de detecção de falhas, deterministicamente alcançar ambas as características de completude e precisão, estando em uma rede assíncrona e não confiável. Como conseqüência, a maioria das aplicações distribuídas tem optado por contornar esta impossibilidade, ao confiar em algoritmos de detecção de falhas que garantam completude deterministicamente, enquanto alcançam eficiência, apenas, probabilisticamente (GUPTA; CHANDRA; GOLDSZMIDT, 2001).

2.2 Considerações finais

Tolerância a falhas é uma questão importante no projeto de sistemas distribuídos, característica pela qual um sistema pode mascarar a ocorrência e a recuperação de falhas.

O objetivo fundamental dos esquemas de tolerância a falhas em sistemas distribuídos paralelos é de que a aplicação seja executada corretamente, mesmo quando falhe qualquer um dos componentes do sistema. Esse requisito é alcançado através de uma sobrecarga na aplicação paralela distribuída.

Para alcançar tolerância a falhas é importante ter mecanismos de proteção, detecção e recuperação de falhas, também é necessário um mecanismo que permita a reconfiguração do sistema para continuar sua execução.

A recuperação em sistemas tolerantes a falhas é alcançada por pontos de verificação periódicos do estado do sistema. A verificação é completamente distribuída, sendo esta uma operação custosa, principalmente no caso de pontos de verificação independentes. Para melhorar o desempenho, muitos sistemas distribuídos combinam pontos de verificação com registro de *log* de mensagens. Este capítulo abordou as diversas opções de tolerância a falhas e os benefícios de cada uma.

Capítulo Três

Estado da arte

3.1 Introdução

A Interface de Troca de Mensagens (MPI) é uma especificação do que uma biblioteca de troca de mensagens deve possuir, e foi construída baseada no consenso do Fórum MPI (MPI, 1994). Essencialmente, essa interface provê rotinas para comunicação e sincronização de aplicações paralelas. Embora a especificação abranja o suporte a muitos recursos para aplicações paralelas, deixa em aberto os problemas relacionados à tolerância a falhas. Entretanto, existem algumas implementações deste padrão que fornecem serviços de tolerância a falhas. A Figura 3.1 apresenta as principais implementações de MPI que oferecem mecanismos de tolerância a falhas.

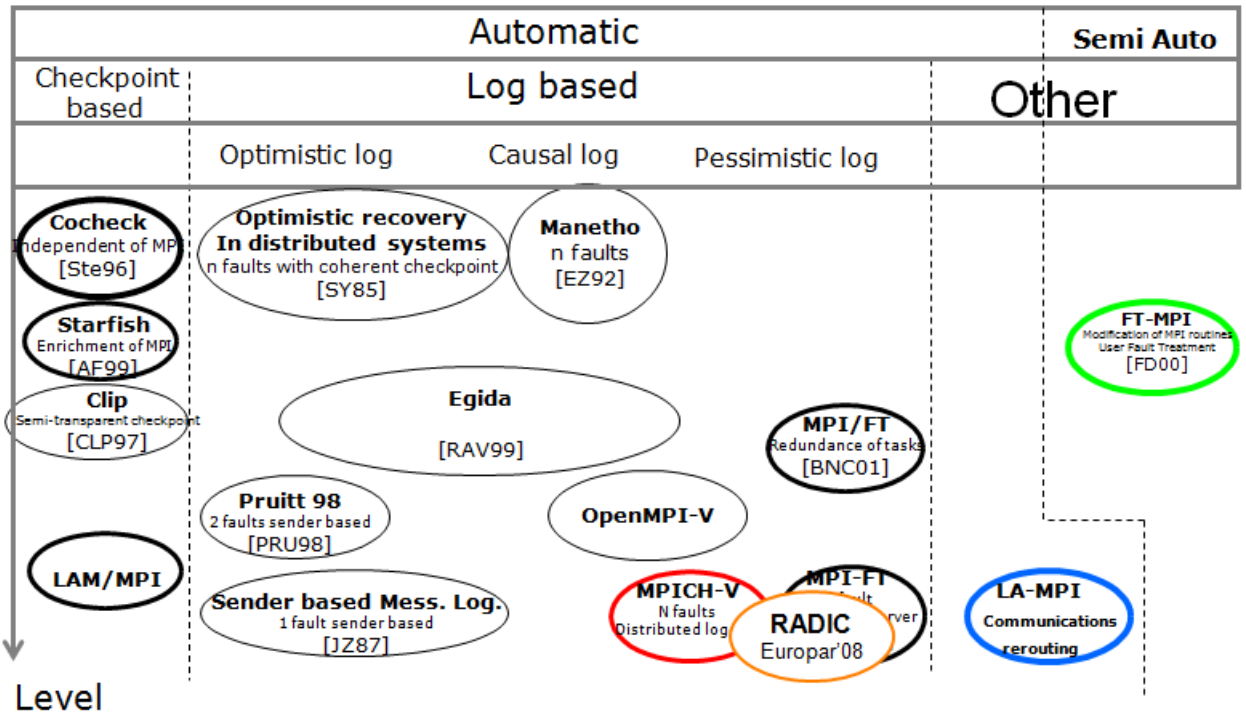

Figura 3.1: Implementações de sistemas de tolerância a falhas em bibliotecas MPI. Fonte: Cappello (2009).

3.2 Implementações de sistemas de tolerância a falhas em bibliotecas MPI

A Tabela 3.2 apresenta os principais sistemas de tolerância a falhas em bibliotecas MPI, desde as implementações de *checkpoint* até as várias formas de *log* de mensagens e outros mecanismos que serão comentados em seguida.

Sistemas	Ano	Serviços			
		Checkpoint		*Log* de mensagens	
		Coordenado	Não-Coordenado	Pessimista	Otimista
MPI-FT	2000			X	X
FT-MPI	2000	X			
STARFISH	2003	X	X		
LAM/MPI	2003	X			
MPI/FT	2004		X	X	
MPICH-V	2006	X	X	X	
RADIC-MPI	2006		X	X	

Tabela 3.1: Sistemas de tolerância a falhas em bibliotecas MPI. Fonte: Autor.

3.2.1 MPI-FT

No MPI-FT (LOUCA et al., 2000), as falhas são detectadas por um observador central. Esse processo observador utiliza um mecanismo que periodicamente checa a existência dos processos. O MPI-FT dispõe de dois recursos para a recuperação desses processos: uma possibilidade é cada processo manter uma cópia de todas as mensagens enviadas; outra abordagem é que todas as comunicações são gravadas pelo processo observador. Caso ocorra uma falha, o processo observador reenvia todas as mensagens do processo faltoso para um processo substituto.

Os processos substitutos são criados dinamicamente ou na inicialização do programa, sendo que, para essa opção, o processo ficará ocioso esperando uma ativação do processo observador. Para utilizar o mecanismo de tolerância a falhas do MPI-FT, o usuário programador deve incluir pontos de verificação de estado, no código da aplicação. Esses pontos de verificação permitem que o processo observador avise, aos processos da aplicação, sobre as falhas ocorridas, e que estes executem as rotinas de recuperação. O MPI-FT admite que o recurso onde o processo observador é instanciado não pode falhar.

3.2.2 FT-MPI

O FT-MPI (FAGG; DONGARRA, 2000) implementa um gerenciador de falhas, através da manipulação da estrutura do *communicator* do padrão MPI. Isso permite à aplicação continuar usando um comunicador com processos falhos, através da eliminação dos processos falhos de seu contexto, ou com a criação dinâmica de processos que substituirão os processos faltosos. FT-MPI não fornece detalhes sobre a estratégia usada para detecção de falhas. Além disso, o desenvolvedor da aplicação é responsável por prover mecanismos de gravação de *checkpoints* e recuperação da aplicação.

3.2.3 STARFISH

Starfish (ADNAN; ROY, 2003) fornece um ambiente de execução para programas MPI que se adaptam às mudanças em um *cluster* de processadores, cujas mudanças são causadas por falhas nos nós. Em cada nó, é executado um *daemon* do Starfish, e os processos de aplicação se registram com o intuito de serem avisados sobre falhas de processos. O Starfish provê duas formas de tolerância a falhas. A primeira consiste na escolha do usuário pelo uso de protocolo coordenado ou não coordenado na gravação de *checkpoints* que, em caso de falhas, são utilizados para reiniciar a aplicação. A segunda é mais dependente da aplicação, visto que, na ocorrência de falhas, os processos sobreviventes são avisados, e, no momento em que ficam cientes da falha, eles repartem o conjunto de dados e continuam sua execução.

3.2.4 LAM/MPI

LAM/MPI (SQUYRES, 2003) utiliza uma biblioteca de gravação de *checkpoints* no nível de sistema BLCR (HARGROVE; DUELL, 2006). Essa biblioteca faz *checkpoint* coordenado, usando o algoritmo de Chandy e Lamport (1985). Além disso, ela pode reiniciar a execução de todos os processos da aplicação. A fim de detectar falhas, os *daemons*, inicializados pelo LAM-MPI em cada nó, trocam mensagens de *heartbeat* no próprio nó.

Quando um nó é considerado faltoso, todos os outros processos são notificados através de um sinal de interrupção. Assim, o desenvolvedor da aplicação, através das funções **lamshrink** e **lamgrow**, pode remover os processos faltosos do contexto do comunicador e adicionar um novo nó onde processos poderão ser instanciados, respectivamente. Ambas as funções são fornecidas pelo LAM-MPI e não fazem parte do padrão MPI.

3.2.5 *MPI/FT*

O MPI/FT (BATCHU et al., 2004) fornece serviços para recuperação e detecção de processos faltosos. O MPI/FT trata falhas, para aplicações que seguem os modelos *Master/Worker*, utilizando o paradigma SPMD, e usa mensagens de *heartbeat* para detectar falhas de processos. Um único processo coordena as funções de detecção e recuperação para toda a aplicação paralela. O MPI/FT faz uso da replicação passiva, para fornecer tolerância a falhas. Dessa maneira, o processo faltoso é substituído por outro, que, por sua vez, reinicia a execução a partir do último *checkpoint* válido. Sempre que um processo extra venha a ser utilizado, o MPI/FT não consegue mais recuperar uma falha para esse processo e, neste caso, a aplicação toda falha.

A consistência de *checkpoints* é estabelecida pela participação de todos os processos em uma operação coletiva de gravação de *checkpoint*, que se comporta essencialmente como uma operação de barreira. No MPI/FT, a aplicação é responsável por realizar a gravação de *checkpoints* e restaurar a execução dos processos, a partir do último *checkpoint* válido.

3.2.6 *MPICH-V*

O projeto MPICH-V (BOUTEILLER et al., 2006) tem sua implementação baseada no MPICH e oferece múltiplos protocolos de recuperação de falhas para aplicações MPI. Atualmente, o projeto MPICH-V oferece cinco protocolos: dois protocolos de *log* pessimista com *checkpoint* não coordenado, um protocolo de *log* causal com *checkpoint* não coordenado e dois protocolos de *checkpoint* coordenado baseado no algoritmo de Chandy e Lamport (1985).

O projeto MPICH-V é composto por um conjunto de componentes, entre os quais, o despachante que, através de mensagens de *heartbeat* enviadas pelos processos da aplicação MPI, detecta a ocorrência de falhas potenciais devido à demora de uma dessas mensagens. Sendo uma falha detectada, o despachante inicia dinamicamente outro processo MPI. Esse processo novo reinicia a execução, alcança o ponto de falha e continua sua execução a partir deste ponto. Os outros processos não ficam cientes da falha ocorrida.

3.2.7 *RADIC-MPI*

O projeto RADIC-MPI (DUARTE; REXACHS; LUQUE, 2006) utiliza processos protetores e observadores, distribuídos ao longo de cada nó de um *cluster*, os quais cooperam entre si para fornecer o serviço de tolerância a falhas. O processo observador é responsável

por gravar o *checkpoint* dos processos presentes em seu nó e transmiti-lo a um processo protetor, presente em outro nó, para que este o armazene em um repositório local. Além disso, os processos observadores são responsáveis por entregar todas as mensagens trocadas entre os processos de aplicação. Dessa forma, eles também salvam essas mensagens e as enviam para os processos protetores.

São utilizadas mensagens de *heartbeat* e gravação de *checkpoint*, através da biblioteca BLCR (*Berkeley Labs Checkpoint/Restart*) (HARGROVE; DUELL, 2006), e *log* pessimista, nos processos de detecção e recuperação de falhas respectivamente. Os processos de aplicação remanescentes não são avisados sobre a falha. Somente o processo protetor ou observador que detectou a falha inicia o procedimento de recuperação. Novos processos são criados dinamicamente em outro nó, os *checkpoints* são lidos e mensagens recebidas pelos processos falhos são reenviadas para os processos substitutos. O projeto RADIC-MPI suporta a ocorrência de múltiplas falhas no mesmo instante, desde que essas falhas não sejam correlacionadas, ou seja, um nó de um observador e o nó onde o seu processo protetor se encontra não podem falhar ao mesmo tempo.

3.3 Considerações finais

Algumas das implementações de sistemas de tolerância a falhas em bibliotecas MPI apresentadas na Seção 3.2 propõe modificações não previstas ao padrão MPI, ou então, deixa a cargo do desenvolvedor da aplicação a responsabilidade de prover os mecanismos de gravação de *checkpoints/restart* da aplicação.

O Open MPI (OPENMPI, 2012) é a combinação de vários projetos MPI já existente, com o objetivo de disponibilizar uma única implementação MPI integrando funcionalidades destes projetos. Atualmente, o Open MPI oferece suporte para *checkpoint* coordenado e possui implementações experimentais para *log* de mensagens pessimista/otimista, tolerância a falhas na comunicação de rede e notificação de erros de comunicadores e processos.

O suporte a *checkpoint* coordenado (HURSEY et al., 2007) é baseado na implementação do LAM-MPI (SQUYRES, 2003). Os *checkpoints* individuais dos processos são realizados através da BLCR (HARGROVE; DUELL, 2006). O Open MPI também possui um mecanismo baseado no LA-MPI (AULWES et al., 2004), segundo o qual o meio de transmissão de dados não é confiável. O DR (*Data Reliability*) (SHIPMAN; GRAHAM; BOSILCA, 2007) descarta e reinicia conexões comprometidas, bem como trata da retransmissão de dados corrompidos ou perdidos, para assegurar a confiabilidade da comunicação.

A pesquisa de Hursey (2010) investigou as implicações de desempenho da implementação de *checkpoints/restart* no Open MPI, também foi desenvolvido um conjunto de interfaces

de programação de aplicações (*APIs*) e opções de linha de comando para o Open MPI que foram projetadas para aumentar a adoção do usuário final e integração de *software* de terceiros.

Tendo em vista que o Open MPI já dispõe de interfaces de programação de aplicações (*APIs*) para *checkpoints/restart*, projetadas para permitir que os usuários programadores de computação de alto desempenho possam usar a solução de *checkpoints/restart* sem conhecer os detalhes sobre sua implementação, desenvolveu-se um ambiente computacinal tolerante a falhas para aplicações paralelas que utilizam o Open MPI. Segue no próximo capítulo a proposta de solução.

Capítulo Quatro

Projeto e implementação do ACTF

4.1 Introdução

O presente trabalho propõe um ambiente computacional tolerante a falhas para aplicações paralelas que utilizam o padrão Open MPI, denominado de **ACTF (Ambiente Computacional Tolerante a Falhas)**. O ambiente dispõe de mecanismos para detectar e tratar falhas da *classe fail* stop em aplicações paralelas, ou seja, o ambiente assume que uma falha ocorre quando existe a interrupção do funcionamento de todo o elemento do nó da computação e não, apenas, da CPU ou memória. O ACTF possibilita, aos processos que compõem o nó faltoso, recuperarem-se e reiniciarem-se automaticamente, a partir dos últimos estados armazenados, preservando todo o processamento realizado até a parada do nó falho.

O ACTF utiliza a técnica de monitoramento de processos paralelos por *heartbeat*, para permitir a continuidade da execução dos referidos processos, sem perda de processamento, aqui é empregada à técnica de *checkpoint/restart*. A Figura 4.1 apresenta os componentes da arquitetura do ACTF.

O ACTF tolera múltiplas falhas e assume que o sistema está totalmente conectado, de tal forma que a falha em um *link* de comunicação não particiona a rede de comunicação. Além disso, não são consideradas as falhas ocorridas na inicialização e na finalização da aplicação paralela, bem como, durante o processo de recuperação e gravação de *checkpoints*. Após uma falha, a aplicação usará os recursos remanescentes para continuar sua execução. O ACTF permite a reativação da aplicação interrompida em decorrência de uma queda do *cluster* de computadores por falta de energia, por exemplo.

4.2 Estrutura dos componentes do ACTF

Os componentes da arquitetura do ACTF, especificados resumidamente a seguir, têm funções específica. A camada que representa a aplicação paralela é composta pelo programa paralelo implementado, utilizando a API do padrão Open MPI; a camada que representa o ACTF é composta de três componentes que cooperam para prover os mecanismos de tolerância a falhas. O componente AP representa a aplicação paralela com seu estado modificado, permitindo a inclusão dos componentes DF, GP e GD.

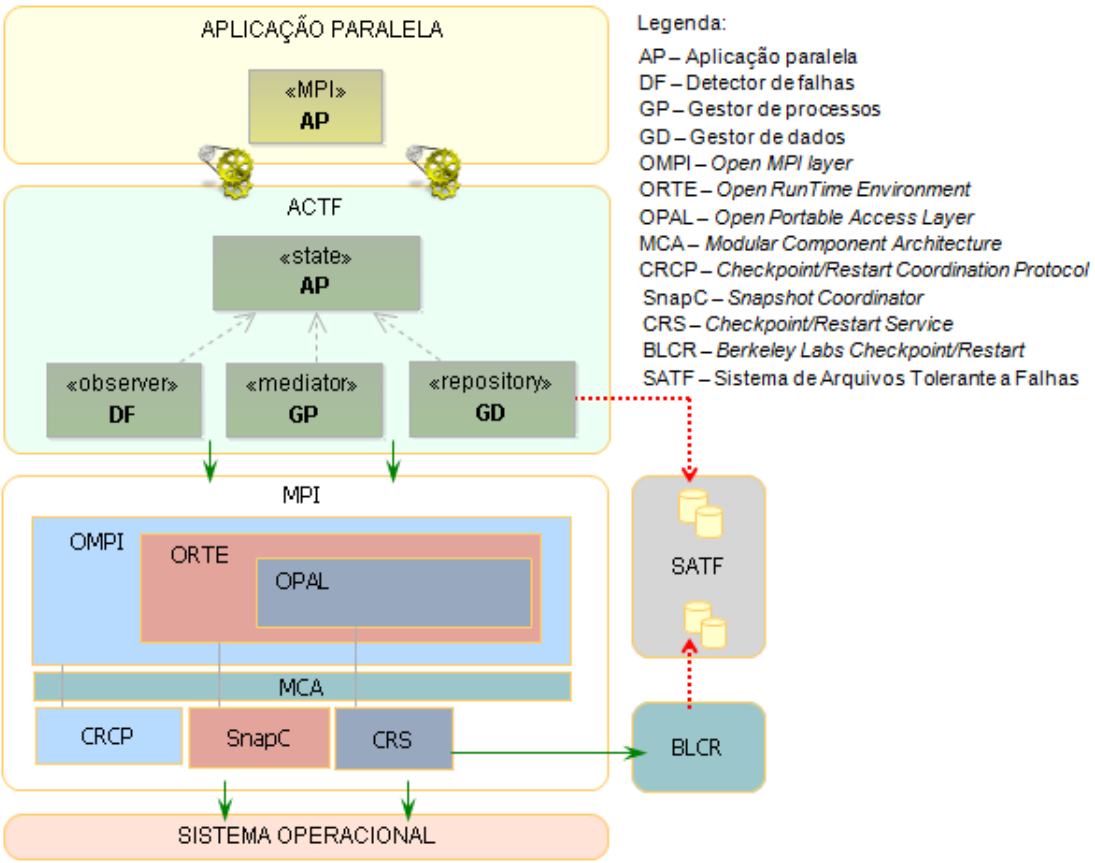

Figura 4.1: Arquitetura do ACTF. Fonte: Autor.

O componente DF representa o detector de falhas, responsável pelo monitoramento dos processos por *heartbeat*. O componente GP representa o gestor de processos, e é responsável por executar o procedimento de reconfiguração da aplicação paralela, em caso de falha. O componente GD representa o gestor de dados, responsável por enviar os pedidos de tomada de *checkpoints* dos processos paralelos.

A camada MPI é composta por três componentes que interagem entre si, para proverem todas as suas funcionalidades. O componente OMPI (*Open MPI layer*) é a camada que provê a interface MPI para as aplicações paralelas. O componente ORTE (*Open Run-Time Environment*) é responsável pela descoberta, mapeamento e alocação dos recursos de processamento disponíveis no ambiente de execução e pelo disparo das tarefas, monitoramento do seu estado e tratamento de erros. O componente OPAL (*Open Portable Access Layer*) é responsável pela abstração das peculiaridades específicas do sistema, oferecendo primitivas para auxiliar na interoperabilidade de diferentes sistemas, com *frameworks* para suporte a *threads*, temporizadores, afinidade de processador, memória e outros recursos dependentes do sistema operacional.

O MCA (*Modular Component Architecture*) permite organizar as funcionalidades do Open MPI em componentes. Essencialmente, o MCA serve para encontrar, carregar e descarre-

gar bibliotecas de vínculo dinâmico, denominadas componentes, que implementam funções definidas pela API interna do Open MPI, estas últimas denominadas *frameworks*.

O suporte a *checkpoint/restart* do padrão Open MPI está organizado em três *frameworks* MCA. São eles: CRCP, SnapC e CRS. O framework CRCP (*Checkpoint/Restart Coordination Protocol*) é responsável pela obtenção do estado global consistente. Essa tarefa envolve o estado dos canais de comunicação da aplicação. O framework CRCP atua diretamente na camada OMPI. O framework SnapC (*Snapshot Coordinator*) atua na camada ORTE e coordena a obtenção do *checkpoint* distribuído. Entre as atividades do SnapC, estão as de iniciar o *checkpoint* de cada processo local, monitorar o progresso do *checkpoint* global e gerenciar os arquivos de contexto gerados. O *framework* CRS (*Checkpoint/Restart Service*) atua na camada OPAL e é responsável pelo *checkpoint/restart* dos processos locais. O CRS implementa uma interface de comunicação com a biblioteca BLCR.

O BLCR é uma implementação robusta de *checkpoint/restart* no nível de sistema. BLCR é implementado como um módulo do *kernel* Linux e uma biblioteca no nível de usuário. A biblioteca permite que aplicações e outras bibliotecas possam interagir com *checkpoint/restart*.

O SATF (Sistema de Arquivos Tolerante a Falhas) é o local de armazenamento confiável que, neste trabalho, utiliza o sistema de arquivos de rede NFS (*Network File System*), onde são registrados os dados para manter o *cluster* operante em caso de falha. Esse componente é imprescindível, em razão de que os dados produzidos serão armazenados e recuperados do mesmo.

Uma representação da colaboração dos elementos que compõem o ambiente ACTF, bem como suas respectivas interações podem ser vista na Figura 4.2 As seções seguintes apresentam um detalhamento sobre a composição e as funcionalidades dos mesmos.

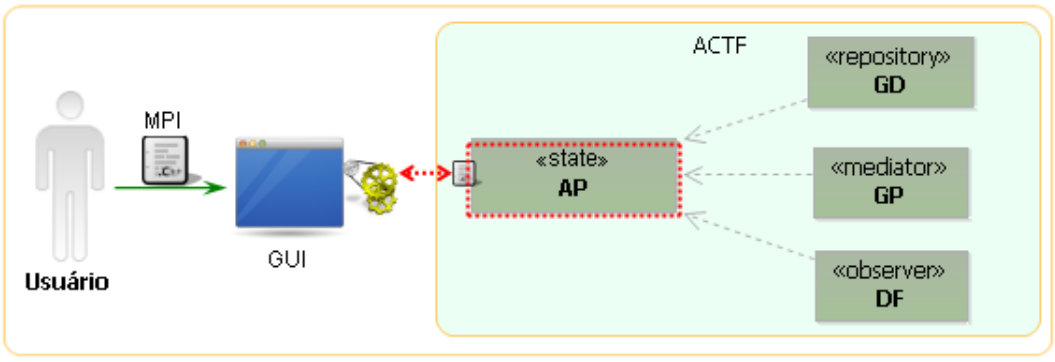

Figura 4.2: Colaboração dos Componentes do ACTF. Fonte: Autor.

4.2.1 A aplicação paralela

A aplicação paralela consiste em uma aplicação desenvolvida, utilizando-se o padrão Open MPI. O ACTF disponibiliza, ao usuário programador, uma interface que permite a inclusão dos componentes (DF, GP, GD) necessários para blindar a aplicação contra o surgimento de falhas. Não há necessidade de o usuário programador adicionar qualquer instrução ao seu código-fonte, para que o mesmo possa emitir as sinalizações necessárias.

O novo estado da aplicação paralela original é alcançado, através da análise de suas propriedades e estrutura, quando esta é submetida a um processo de transformação, disponibilizado por um componente de interface (GUI) do ACTF. Essa transformação envolve a realização dos *includes* necessários no cabeçalho do programa, em seguida, logo após a função **MPI_Init()**, são inseridas as instruções necessárias para o controle dos processos paralelos. Antes da função **MPI_Finalize()**, são inseridas instruções, para finalizar as *threads* criadas pelos componentes (DF, GP, GD). A Figura 4.3 apresenta os componentes do subsistema de transformação do novo estado da aplicação paralela.

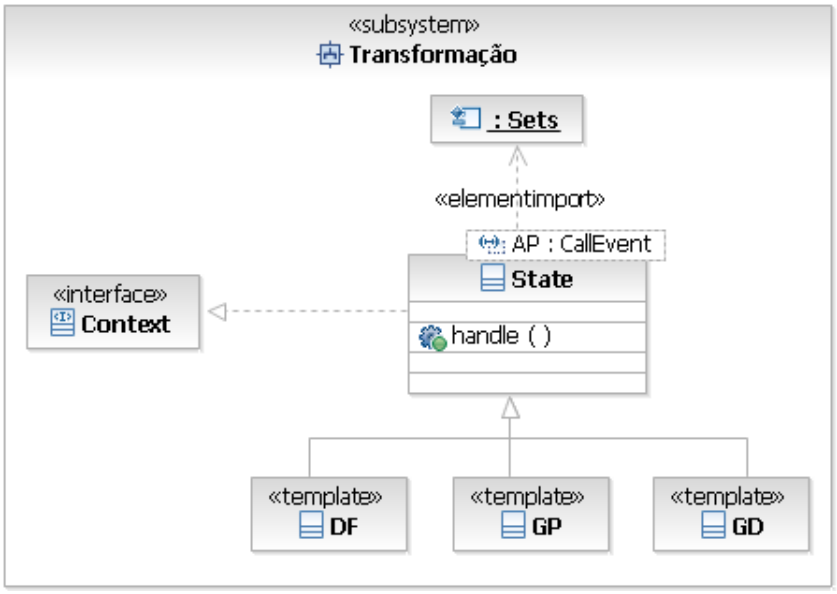

Figura 4.3: Subsistema de Transformação. Fonte: Autor.

O componente *Context* é a interface a ser usada para a transformação do código enviado pelo usuário programador. *Context* mantém uma instância do objeto, que implementa a chamada para a transformação do estado da aplicação paralela.

State é o componente que define o comportamento, funciona como um *template* para criação de novos estados de um determinado contexto. O novo estado da aplicação paralela é alcançado, através da inserção dos componentes DF, GP e GD.

Essa estratégia de transformação permite a inserção de novos comportamentos, de maneira rápida e transparente, para a aplicação paralela, bastando, apenas, a inserção de novos componentes *templates*, e esses devem agregar valor, realizando o componente *State*.

Sets é o componente utilizado pelo *State*, para armazenar o valor dos atributos, que permitem a configuração do ambiente. Os atributos, que podem ser modificados pelo usuário programador, são: *timeout* de inspeção dos processos paralelos e o *timeout* para a tomada dos *checkpoints*. Esses atributos devem ser ajustados, conforme o ambiente onde este mecanismo será empregado. A fixação do valor desses parâmetros implica em um *trade-off* entre corretude e eficiência.

4.2.2 O gestor de dados

O GD é o componente responsável por recepcionar as requisições de tomadas de *checkpoints* dos processos paralelos, além de manter os arquivos de configuração do *cluster* de computadores. Esse componente está replicado em todos os processos MPI, porém só é ativado no processo com o *rank* igual a zero. O componente GD utiliza à variável **checkpointTime**, para controlar a periodicidade de tomada de *checkpoints* dos processos paralelos. Essa variável assume o valor padrão de 90 segundos, caso não seja informado o valor pelo usuário programador no momento da compilação do programa MPI. O valor da variável **checkpointTime** é atualizado a cada solicitação de *checkpoint*, ou seja, o valor é recalculado, utilizando o tempo médio gasto no *checkpoint*, adicionado ao valor inicial solicitado pelo usuário programador.

Os dados de configuração são passados para o GD no momento da alocação desse componente na memória do nó em que o mesmo está executando. A Figura 4.4 ilustra o subsistema de armazenamento dos dados de configuração do *cluster*.

4.2.2.1 Mecanismo de armazenamento

Logo após o componente GD ser alocado em memória, o procedimento **init()** é chamado. Esse procedimento faz com que o GD use os serviços do componente DAO (*Data Access Object*), para tornar os dados de configuração do *cluster* persistentes, ou seja, armazena os dados em disco para posterior recuperação. O componente DAO mantém duas entidades importantes para a restauração do sistema, em caso de falha. A entidade *Resource* é composta de um atributo primitivo do tipo inteiro, denominado *status*, podendo assumir dois valores: 0 (zero), indicando que o *cluster* está operante, e 1 (um), indicando que o *cluster* terminou a computação sem falhas. O atributo *status* é inicializado com o valor

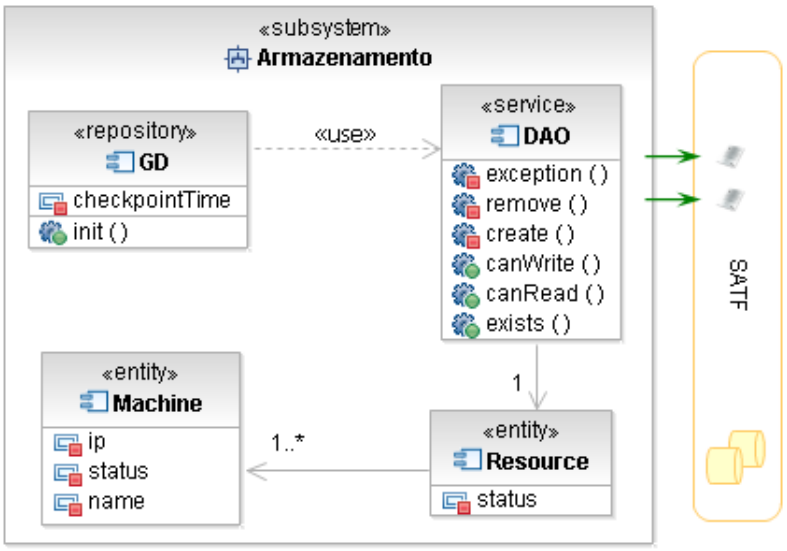

Figura 4.4: Subsistema de Armazenamento. Fonte: Autor.

default 0 (zero). Além do *status*, a entidade *Resource* tem uma lista de entidades do tipo *Machine*, que é composta pelos atributos: *ip*, que representa o endereço de rede da máquina; o *status*, que representa o estado da máquina, podendo assumir os valores: 0 (zero), inoperante, ou 1 (um), operante, e o atributo *name*, que representa o nome da máquina. Pelo menos uma máquina deve existir para que seja possível a execução dos processos MPI no *cluster* de computadores.

Quando o processo MPI de *rank* zero termina a sua execução, o componente GD é acionado pela aplicação paralela, para remover as estruturas criadas no SATF, no início da computação. Esse procedimento é realizado através da função **remove()**.

O mecanismo de armazenamento permite a recuperação da computação paralela distribuída, em caso de parada total do *cluster* de computadores. Os passos do procedimento de recuperação, em caso de parada total do *cluster*, são listados a seguir:

1. O GD recebe uma mensagem de inicialização, logo em seguida, o procedimento **init()** é executado, instanciando o componente DAO.
2. O DAO chama o procedimento **exists()**, e este verifica se as estruturas de dados (*Resource* e *Machine*) foram criadas no SATF. Em caso de parada total do *cluster*, os arquivos referentes às estruturas estão mantidos no SAFT (Sistema de Arquivos Tolerante a Falhas).
   (a) Caso as estruturas não existam, o DAO torna as estruturas persistentes, através do procedimento **create()**. Logo em seguida, o procedimento **canWrite()** é chamado para atualizar os valores dos atributos das entidade *Resource* e *Machine*. Nesse caso, o passo 3 não será executado.

3. O DAO chama o procedimento **canRead()**, para obter o valor do atributo status da entidade persistente *Resource*.

   (a) Caso o valor seja 0 (zero), o DAO assume que houve parada total do *cluster*, antes do término da computação. Nesse momento, o DAO chama o procedimento **exception()**, que lança uma exceção a ser interceptada pelo GD que, por sua vez, irá notificar o componente GP, para que este realize o procedimento de *restart*.

4.2.2.2 Mecanismo de checkpoint

O mecanismo de *checkpoint* do ACTF consiste na criação de um arquivo de descrição de um processo em execução, o qual pode ser utilizado, para reconstruir o processo, em caso de falha. Este arquivo contém uma imagem do estado de execução do processo em um dado instante de tempo. Isto permite que o processo possa continuar sua execução, a partir do ponto onde o *checkpoint* foi realizado.

Em uma aplicação paralela MPI, os processos cooperam através de trocas de mensagens. A execução desses processos pode ser vista como uma sequência de envio e recebimento de mensagens. Cada evento de envio ou de recebimento de mensagens muda o estado do processo MPI. Dessa forma, o conjunto formado pelos estados locais de todos os processos e seus canais de comunicação representa o estado global da aplicação (SRIRAM et al., 2005). Qualquer *checkpoint* global consistente pode ser usado para reiniciar a aplicação de forma correta. Dizemos que temos um *checkpoint* global consistente, quando o estado formado pelos *checkpoints* locais de cada processo, mais o canal de comunicação é consistente, porntanto não poderá existir mensagens que ultrapassem as linhas de recuperação. A Figura 4.5 ilustra a formação do estado global em uma aplicação, formada por dois processos (p0 e p1).

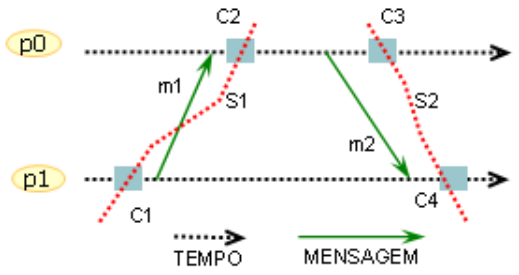

Figura 4.5: Obtenção do estado global consistente. Fonte: Autor.

A Figura 4.5 mostra a troca de mensagens entre dois processos distintos, p0 e p1. O *checkpoint* C1 precede m1 C2, de modo que o *checkpoint* global S1, formado por C1 e C2, não é consistente. Já o *checkpoint* global S2, formado por C3 e C4, é consistente, pois

não existem relações de precedência de mensagem. A Figura 4.6 ilustra o procedimento de *checkpoint* em uma aplicação, que utiliza o mecanismo do Open MPI em cooperação com o ACTF.

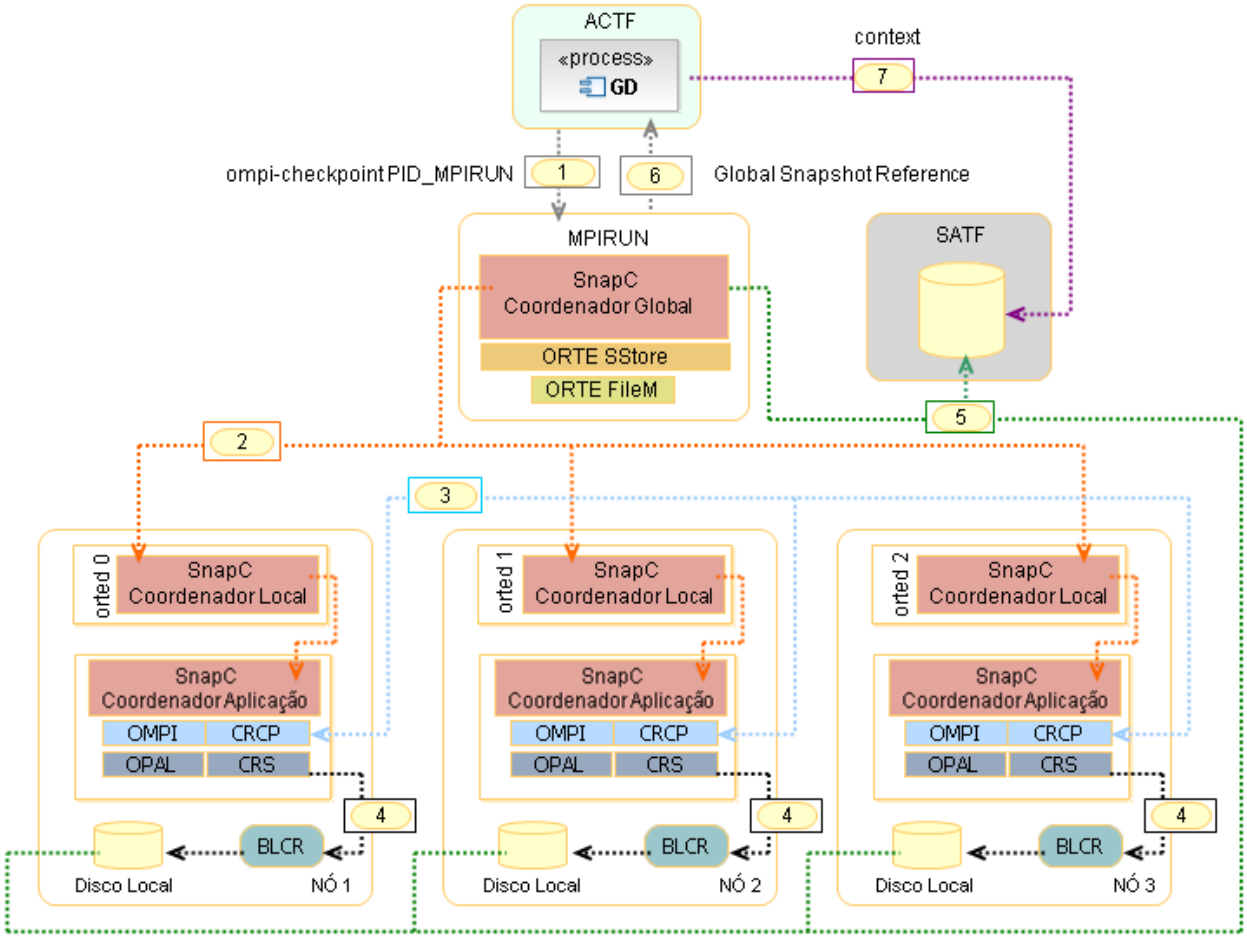

Figura 4.6: Mecanismo de checkpoint. Fonte: Autor.

O ACTF utiliza o suporte nativo do Open MPI para a tomada dos *checkpoints*, ou seja, uma vez solicitado um *checkpoint*, um protocolo coordenado é iniciado, a fim de esvaziar os canais de comunicação, produzindo um estado global consistente, para, então, salvar o contexto dos processos MPI. O GD, de maneira automática, realiza periodicamente o procedimento de *checkpoint* dos processos MPI, através de chamadas à interface do **MPIRUN**. O padrão Open MPI, através do componente ORTE SnapC, implementa uma estratégia de coordenação centralizada. Tratam-se de três coordenadores: um coordenador global, um conjunto de coordenadores locais e um conjunto de coordenadores de aplicação.

O coordenador global é responsável por interagir com as requisições de linha de comando, geração de uma referência global, pela geração dos arquivos remotos, e armazenamento em mídia confiável, através do *framework* ORTE SStore e, pelo monitoramento das solicitações dos *checkpoints*. O coordenador local faz parte da camada ORTE, tendo, cada

nó, um representante (**orted**) do coordenador local. Cada coordenador local coopera com um coordenador global, para iniciar o ponto consistente. O coordenador de aplicação faz parte de cada processo do sistema distribuído, e é responsável por iniciar o *checkpoint* de um único processo.

O procedimento de *checkpoint* inicia-se com uma solicitação realizada pelo GD, através do comando **ompi-checkpoint PID_MPIRUN**. O ponto de entrada da solicitação é o **MPIRUN**. Os passos para a execução desse procedimento são listados a seguir:

1. O GD realiza uma chamada à interface **MPIRUN**, através da chamada **shell$ ompi-checkpoint PID_MPIRUN**. Esse comando produz um arquivo de contexto para o **MPIRUN** e outro, para cada um dos processos paralelos, que compõem a aplicação. A interface **MPIRUN** recebe a solicitação de *checkpoint* e propaga-a para todos os nós da aplicação.
2. Em cada nó do *cluster* de computadores, as camadas OPAL, ORTE e OMPI são informadas da solicitação, e inicia-se a preparação para o *checkpoint*.
3. O *framework* OMPI CRCP negocia um estado global consistente com todos os processos.
4. O *framework* OPAL CRS utiliza o componente BLCR, para salvar o contexto dos processos locais.
5. O *framework* ORTE SnapC centraliza todos os arquivos de contexto e as informações necessárias para a recuperação da execução em um sistema de arquivos tolerante a falhas (SATF).
6. O *framework* OMPI CRCP normaliza as operações, e os processos MPI continuam sua execução. Por fim, o GD recebe uma referência do contexto de execução da aplicação.
7. O GD armazena o arquivo de contexto do **MPIRUN** no SATF.

A partir da referência retornada no *checkpoint*, é possível recuperar a execução da aplicação. Esse mecanismo será detalhado nas próximas seções.

4.2.3 O detector de falhas

O monitoramento dos processos no ACTF é realizado através da estratégia *push*, gerando apenas uma troca de mensagem entre o processo monitorado e o processo monitor. O

mecanismo de monitoramento é descentralizado, tendo um processo DF criado para cada nó do *cluster* de computadores. A Figura 4.7 apresenta a estratégia de monitoramento implementada no ACTF.

Os processos detectores de falhas formam a estrutura de uma lista encadeada circular, tendo, em um processo DF, a referência para o próximo elemento da lista, e o último elemento, a referência para o primeiro, formando um ciclo de elementos DF. Cada processo DF mantém uma referência da lista circular na memória principal e em disco local do nó em que reside. Isso permite conhecer as dependências entre todos os processos.

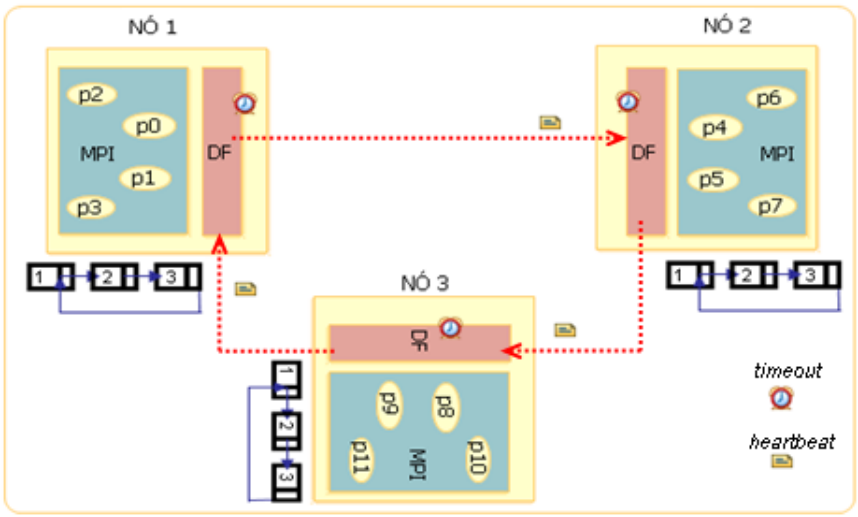

Figura 4.7: Monitoramento descentralizado do ACTF. Fonte: Autor.

Um processo DF monitor (NÓ 2) aguarda o envio de pacotes periódicos (*heartbeat*) procedentes do processo DF monitorado (NÓ 1). Quando o DF monitor detectar que um DF monitorado não enviou o pacote dentro do tempo estabelecido (*timeout*), então o mecanismo assume que o DF monitorado está indisponível, e este será excluído logicamente do *cluster* de computadores.

O componente DF utiliza à variável **timeout**, para representar o tempo máximo de espera por uma mensagem de *heartbeat*. O valor da variável **timeout** é formado pela multiplicação de três vezes o valor do *heartbeat*. O valor do tempo de inspeção é informado pelo usuário e é atribuído à variável **heartbeat**. Essa variável assume o valor padrão de 45 segundos, caso não seja informado pelo usuário. A Figura 4.8 ilustra o funcionamento do mecanismo de detecção de falha de um determinado nó faltoso.

Quando um processo DF monitor detecta a falta de um processo monitorado, ele envia uma mensagem para todos os outros processos detectores, informando o nó que falhou. Em seguida, todos os processos detectores reestruturam a lista encadeada circular, formando um novo ciclo de elementos DF. O valor do *timeout* para o *heartbeat* de cada

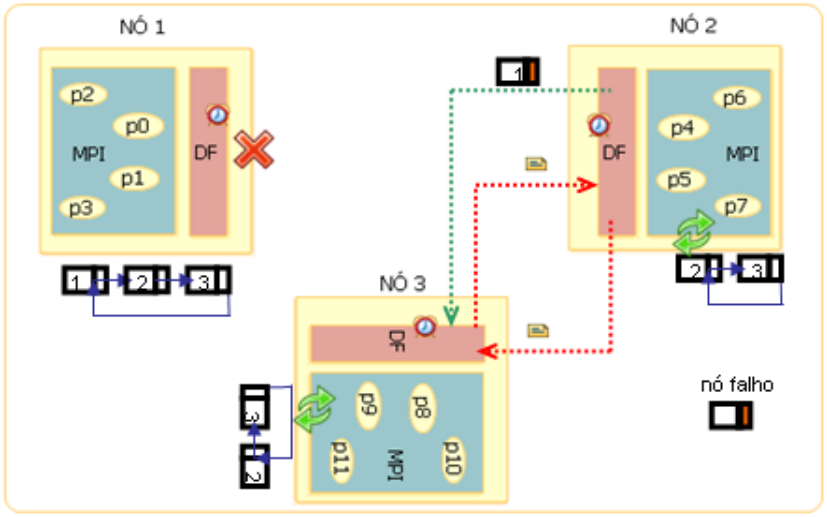

Figura 4.8: Detecção de falha de um determinado nó. Fonte: Autor.

processo DF também é reajustado. O componente DF não tem a responsabilidade de recompor o sistema: esse componente apenas detecta falhas nos nós de execução do *cluster* de computadores. O mecanismo de reconfiguração do sistema é de responsabilidade do componente GP.

4.2.4 O gestor de processos

O GP é o componente que tem a responsabilidade de executar o procedimento de *restart* da aplicação paralela. Este procedimento consiste em recuperar os últimos estados locais dos processos paralelos, armazenados em seus respectivos arquivos de contexto, bem como, o estado global da aplicação. Antes da recuperação, deve ser feita uma interrupção geral da aplicação paralela.

O GP, através de uma interface, observa toda mudança de estado, de modo que, se a estrutura da lista encadeada mantida pelo DF sofrer alguma modificação de estado, o GP será notificado automaticamente. O DF tem a referência para um observador que define uma interface. Para o componente GP ser notificado, sempre que um DF modificar o estado de sua estrutura *list* que armazena a lista de elementos DF, o procedimento **notify()** será acionado, para notificar, ao seu observador, sobre a mudança de estado. A Figura 4.9 ilustra o mecanismo de notificação. O componente GP, quando notificado sobre a atualização do estado de um DF, realiza o procedimento de reconfiguração do sistema, através do procedimento **configure()**.

Para que seja possível a recuperação da aplicação paralela, o GP recupera o arquivo de contexto armazenado no SATF, a partir da referência gerada no último *checkpoint*

realizado pelo GD. A partir dessa referência, é possível recuperar a execução da aplicação (*restart*).

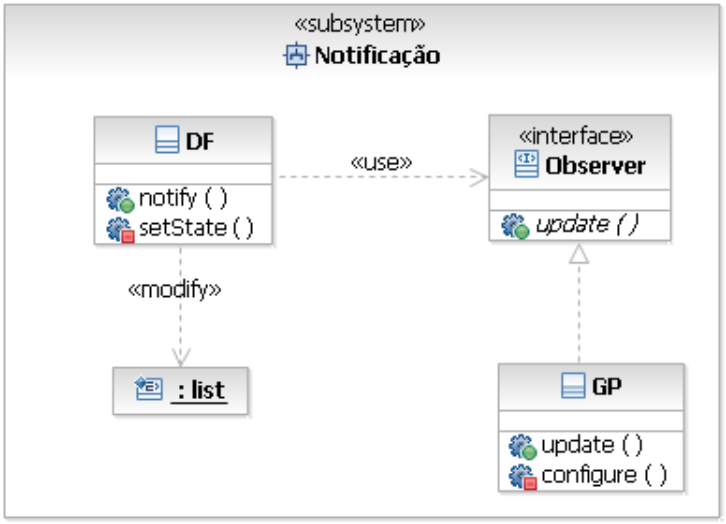

Figura 4.9: Subsistema de notificação. Fonte: Autor.

O procedimento de *restart* inicia-se com uma solicitação realizada pelo GP, através do comando **ompi-restart GLOBAL_SNAPSHOT_REF**. O ponto de entrada da solicitação é o **MPIRUN**. Os passos para a execução desse procedimento são listados a seguir:

1. O GP realiza uma chamada, à interface **MPIRUN**, através da chamada **shell$ ompi-restart GLOBAL_SNAPSHOT_REF**. A interface **MPIRUN** inicia a sua execução, através da referência **GLOBAL_SNAPSHOT_REF**, solicitando a recuperação de todos os processos da aplicação.
2. O framework ORTE SnapC solicita a transferência dos arquivos de contexto do SATF para os nós da aplicação.
3. O *framework* OPAL CRS recupera a execução de cada processo, a partir de seus arquivos de contexto.
4. O *framework* OMPI CRCP informa, aos componentes de comunicação da camada OMPI, que se trata de uma operação de *restart*, e os canais de comunicação entre os processos são recriados.

Após esses passos, a aplicação é reiniciada em um novo conjunto de nós, formado pelas máquinas restantes do *cluster* de computadores.

4.3 Implementação

Os componentes de serviços GD, DF e GP do ACTF foram implementados na linguagem C++, utilizando-se a técnica de programação, baseada na definição de tipos estruturados de dados. A ideia central é encapsular, de quem usa os serviços, os tipos e a forma concreta como eles foram implementados. O usuário programador utiliza os serviços de forma abstrata, ou seja, baseado, apenas, nas funcionalidades oferecidas pelas interfaces dos serviços.

Todas as camadas do ACTF são compostas por um ou mais pacotes. Deste modo, a estrutura de pacotes do ACTF segue o formato presente no diagrama da Figura 4.10.

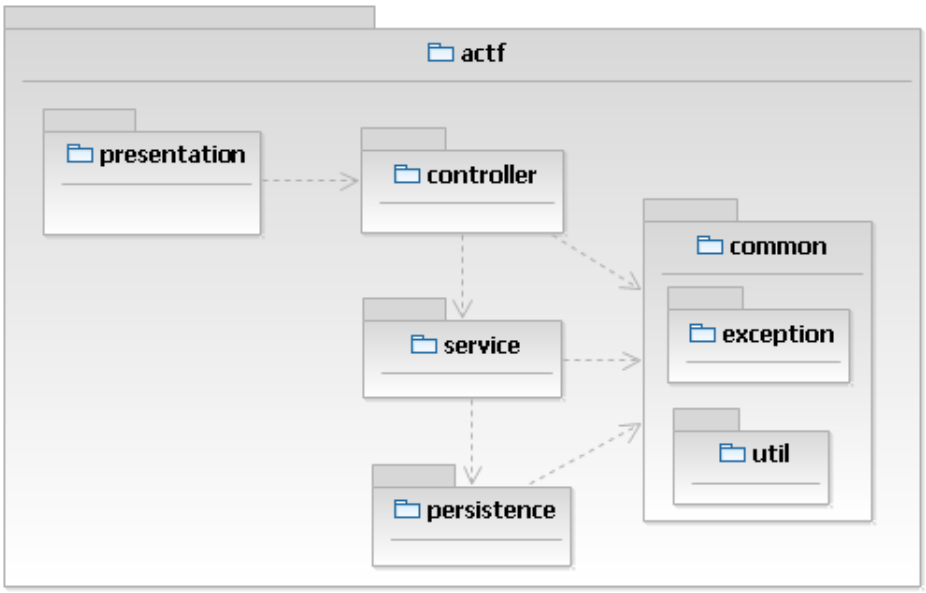

Figura 4.10: Pacotes do ACTF. Fonte: Autor.

No pacote presentation, conforme Figura 4.10, são armazenadas as classes de UI (User Interface) responsáveis pela interação do sistema com o usuário programador. Este pacote comunica-se com os elementos contidos no pacote controller, neste local, ficam os controladores da camada de apresentação. O pacote service provê classes referentes à lógica da aplicação, gerência das classes de negócio e controle transacional, sendo independente da camada de apresentação. Este pacote comunica-se com os elementos contidos no pacote persistence. O pacote persistence é responsável pelo acesso à tecnologia de persistência da aplicação.

O pacote common representa uma camada de suporte para a aplicação, sendo acessível aos elementos dos pacotes controller, service e persistence. Trata-se de um conjunto de classes utilitárias da aplicação.

Capítulo Cinco

Avaliação e resultados experimentais

Para a avaliação do ACTF, foi utilizada a técnica de injeção de falhas por *software*, por se tratar de um método eficiente para a validação de sistemas (JULIANO; WEBER, 2006). A abordagem visa à emulação de falhas de *hardware*, sem comprometer fisicamente o ambiente, e a análise do comportamento do sistema na presença destas falhas. Os experimentos foram realizados no laboratório de HPC (*High Performance Computing*) do Centro Integrado de Manufatura e Tecnologia (SENAI CIMATEC). O ambiente foi composto por um *cluster* de computadores, formado por 6 *blades* HP ProLiant DL120 G6 Quad core Intel® Xeon® HP X3450 (2.67GHz, 95W, 8MB, 1333, HT, Turbo), utilizando-se do sistema operacional Ubuntu/Linux 64 bit, Open MPI, versão 1.7. A rede de conexão utilizada segue o padrão Gigabit Ethernet.

Com o objetivo de comprovar a eficiência do ACTF, experimentos foram realizados utilizando-se seis nós do cluster de computadores, sendo cinco deles utilizados como nós de computação, e um, como nó de armazenamento, conforme ilustra a Figura 5.1.

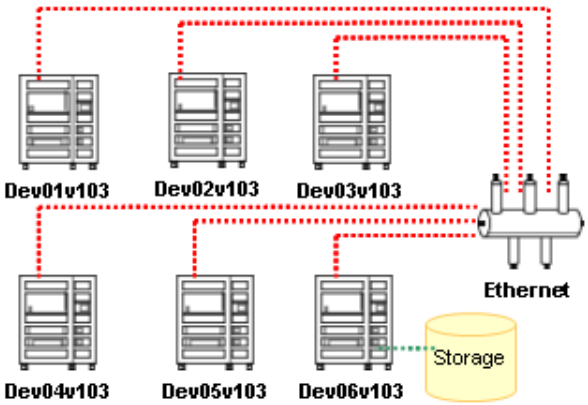

Figura 5.1: Representação do *cluster* de computadores utilizado nos testes. Fonte: Autor.

A aplicação paralela, utilizada para a validação do ACTF, foi o programa de processamento de imagens digitais. O processamento de imagens digitais vem sendo usado, para identificar e reconhecer padrões, e facilitar a interpretação visual (CRóSTA, 1993), sendo a convolução uma técnica de filtragem de frequência, que atua no domínio espacial da imagem, a fim de se produzir o efeito que se deseja (GONZALEZ R., 2000). A convolução é uma técnica, que atua sobre o domínio espacial, por meio de uma matriz, denominada de janela ou máscara, aplicada sobre todos os *pixels* da imagem original, a fim de produzir outra imagem de saída. O programa paralelo implementado neste trabalho utiliza a técnica de convolução, proposta por Gonzalez R. (2000), para gerar o índice de fragmentação de uma imagem.

Turner (1989) apresenta o índice de fragmentação, para medir o grau de variabilidade da paisagem e revelar as possíveis influências da atividade humana sobre a mesma. O índice de fragmentação pode ser adotado como uma medida local de textura, com valores no domínio entre zero e um, sendo calculado para cada *pixel* da imagem, por meio da aplicação de uma convolução específica na região de vizinhança do *pixel*. O índice de fragmentação é um método concebido para representar a variabilidade interna da paisagem, no qual a textura é analisada segundo a frequência de diferentes categorias de paisagem presentes em uma matriz quadrada de *pixels* da imagem. O índice de fragmentação (F) é definido pela Equação 5.1:

$$F = (n - 1)/(c - 1) \tag{5.1}$$

Onde (n) é o número de *pixels* com diferentes atributos (níveis de cinza), presentes em uma janela de dimensão igual a (c) (no caso, janela 3X3, então (c) é igual a 9).

O índice de fragmentação da paisagem é uma medida local de textura e indica que, se todo e qualquer atributo não se repete na janela, o valor calculado para o *pixel* central será máximo (*F=1*). Caso contrário, se todos os *pixels* da janela tiverem o mesmo atributo (*F=0*), não ocorre variabilidade espacial na região analisada de 3x3 *pixels*.

O programa foi projetado, utilizando-se o paradigma SPMD (*Single Program Multiple Data*) e implementado sob o esquema *Master/Worker*. No modelo SPMD, o mesmo programa é executado em diferentes máquinas, entretanto esses programas manipulam um conjunto de dados diferentes. No esquema *Master/Worker*, o *Master* é o processo responsável por decompor o problema em diversas tarefas menores e distribui-las entre os *Workers*. O *Worker* é o processo que recebe os trabalhos, processa-os e devolve-os para o *Master*, que gerencia, organiza e controla os dados processados. A comunicação entre os processos (*Master/Worker*) é feita através da troca de mensagens, realizadas por chamadas explícitas às rotinas da biblioteca Open MPI. A Figura 5.2 apresenta a estratégia de distribuição adotada pelo programa paralelo.

O processo *Master* realiza a leitura da imagem, em seguida, particiona a imagem proporcionalmente ao número de *Workes* participativos na computação paralela distribuída. O processo de particionamento é realizado através da dimensão da imagem, ou seja, da leitura das linhas pelas colunas, gerando uma matriz. As linhas da matriz são enviadas para os *Workes*, através das primitivas de comunicação do Open MPI. Após receber o seu respectivo segmento da imagem, cada *Worker* executa o algoritmo, para gerar o índice de fragmentação de uma imagem, e envia o segmento resultante da convolução ao processo *Master*, que concatena adequadamente os segmentos gerados individualmente e gera a imagem de saída completa.

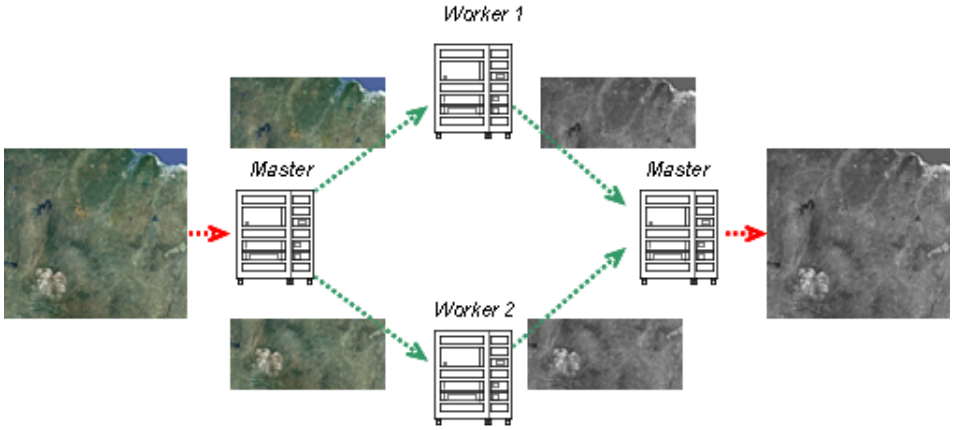

Figura 5.2: Representação da distribuição de tarefas. Fonte: Autor.

Para utilizar o ACTF, é necessário que a aplicação paralela, implementada no padrão Open MPI, seja submetida a um processo de transformação, como mencionado na Seção 4.2.1. O ACTF disponibiliza uma interface (GUI), que permite, ao usuário programador, compilar o seu código fonte, inserindo os componentes DF, GD e GP, com o objetivo de tornar a aplicação paralela tolerante a falhas.

A Tabela 5.1 apresenta os experimentos utilizados para a validação do ACTF. Os resultados dos experimentos foram obtidos pela média de três execuções, cada experimento foi executado cinco vezes, onde o melhor e o pior valor foram retirados.

Como pode ser visto na Tabela 5.1, o experimento "Influência dos parâmetros do DF" o componente DF foi configurado com valores para o *heartbeat* iguais a 15, 30, 45 e 60 segundos, e o *cluster*, configurado com 4, 8, 12 e 20 processos. Esse experimento tem por objetivo avaliar a influência do componente DF no tempo final da aplicação paralela, esse experimento também avalia o tempo de detecção de falhas.

Experimento	Componente				
Influência dos parâmetros do DF	*heartbeat (s)*	processos			
	15	4	8	12	20
	30	4	8	12	20
	45	4	8	12	20
	60	4	8	12	20
Influência dos parâmetros do GD	*checkpointTime (s)*	processos			
	30	4	8	12	20
	60	4	8	12	20
	90	4	8	12	20
	120	4	8	12	20
Sobrecarga imposta pelo ACTF	*heartbeat (s)*	60			
	checkpointTime (s)	90			
	processos	20			

Tabela 5.1: Experimentos utilizados para validação do ACTF. Fonte: Autor.

O experimento "Influência dos parâmetros do GD" visa avaliar a influência do mecanismo de *checkpoint* do componente GD no tempo final da aplicação paralela. Nesse experimento, o componente GD foi configurado com valores para o *checkpointTime* iguais a 30, 60, 90 e 120 segundos, o *cluster* foi configurado com 4, 8, 12 e 20 processos.

Já o experimento "Sobrecarga imposta pelo ACTF" tem por objetivo avaliar a sobrecarga na aplicação paralela, provocada pelo uso do mecanismo de tolerância a falhas, implementado no ACTF. O tempo para a tomada de *checkpoint* dos processos paralelos foi configurado com o valor de 90 segundos, e o *heartbeat* foi configurado com o valor de 60 segundos. Foram utilizados 20 processos em cada execução da aplicação paralela.

O ACTF disponibiliza uma interface para a execução da aplicação paralela. Essa interface permite a cópia e compilação do arquivo fonte nas máquinas participativas do *cluster* de computadores. A Figura 5.3 apresenta a interface para execução da aplicação paralela.

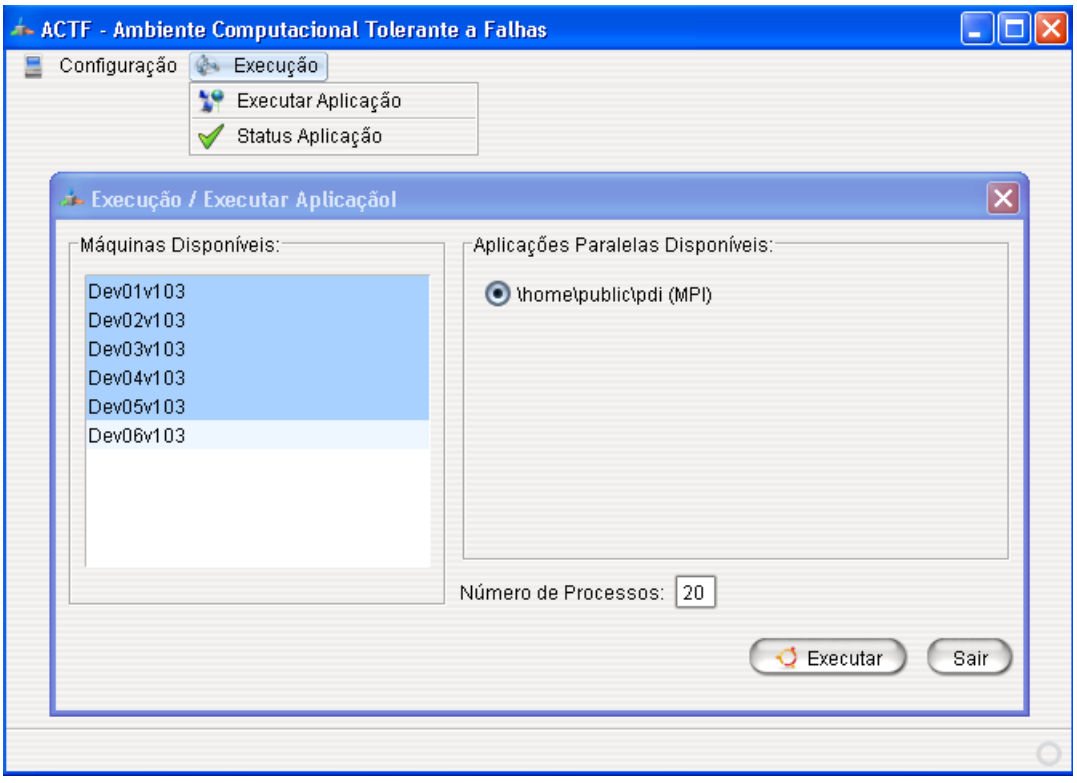

Figura 5.3: Interface de execução. Fonte: Autor.

O usuário programador seleciona as máquinas, que representam os nós disponíveis no *cluster* de computadores, escolhe a aplicação paralela, que deseja executar no ACTF, em seguida, informa o número de processos a serem criados pelo MPI. A partir desses passos, é liberada a opção "Executar", que permite a cópia do programa selecionado para a pasta padrão "\home\public" das máquinas selecionadas. Após a cópia, o ACTF realiza o processo de compilação em cada nó do *cluster* e, em seguida, realiza a chamada a interface **MPIRUN**, através do comando **mpirun -hostfile hostACTF -np 20 pdi**.

O arquivo hostACTF é formado a partir das máquinas selecionadas, para permitir a execução distribuída dos processos MPI.

5.1 Influência dos parâmetros do mecanismo de detecção de falhas

A decisão de quando a informação fornecida por um detector de falhas deve ser considerada verdadeira é de responsabilidade da aplicação (STELLING et al., 1998). Esta decisão envolve custos ao programa, para realizar uma ação de reconfiguração do ambiente. O componente DF utiliza à variável *timeout*, para representar o tempo máximo de espera por uma mensagem de *heartbeat*. A parametrização dessa variável com um valor muito longo implicaria em uma demora na detecção de falhas, o que comprometeria a eficiência do DF. Determinar bons valores para esse parâmetro se torna um desafio. Nesse sentido, foram realizados experimentos em que alguns valores para o *heartbeat* foram testados, a fim de avaliar o impacto dessa escolha no tempo da aplicação.

5.1.1 Influência do componente DF em um cenário livre de falhas

Esse experimento visa avaliar a influência do componente DF em um cenário de execução livre de falhas. Nesse experimento, o componente DF foi configurado com valores para o *heartbeat* iguais a 15, 30, 45 e 60 segundos, e a aplicação, configurada com 4, 8, 12 e 20 processos. A Figura 5.4 apresenta o tempo de execução da aplicação, apenas, com o componente DF ativo.

Cada coluna da Figura 5.4 representa o tempo total de execução da aplicação paralela, em determinada situação. A coluna "SDF" representa a execução da aplicação sem o DF ativo, ou seja, trata-se de uma execução normal da aplicação original, sem a injeção de qualquer componente do ACTF. A coluna "heartbeat=15" representa a execução da aplicação paralela, com o valor da variável *heartbeat* do DF configurada para que, a cada 15 segundos, sejam enviadas as mensagens de confirmação de operação dos processos monitorados para os processos monitores. O mesmo ocorre para as colunas "heartbeat=30", "heartbeat=45" e "heartbeat=60", tendo apenas a periodicidade de envio das mensagens variada em função dos valores do *heartbeat*.

Com os valores obtidos nesse experimento, pode-se observar que, quanto maior o valor do *heartbeat*, menor é a influência do mecanismo de detecção de falhas no tempo final do processamento da aplicação paralela.

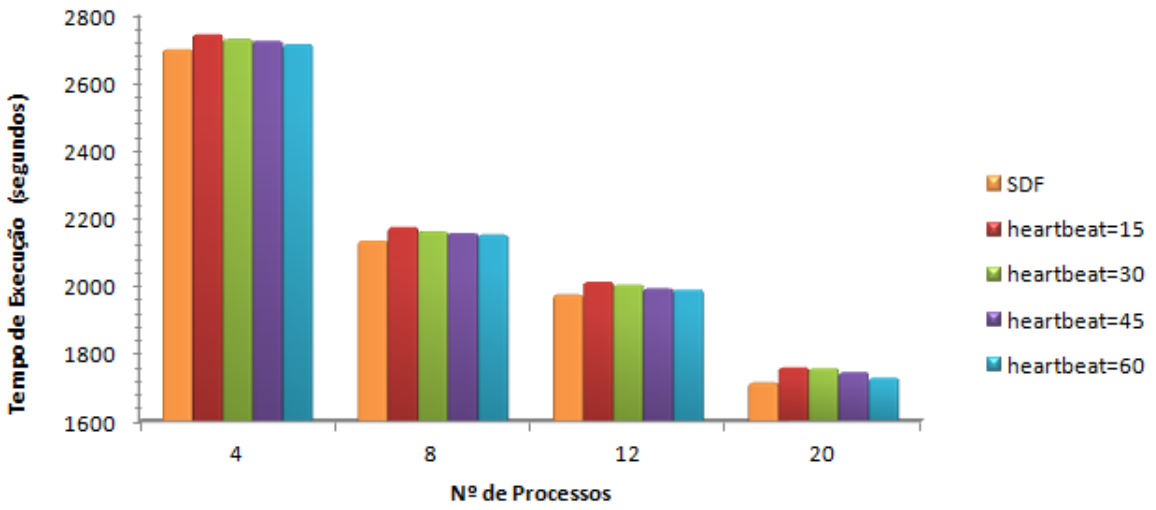

Figura 5.4: Tempo de Execução em um cenário livre de falhas com o DF ativo. Fonte: Autor.

A Tabela 5.2 apresenta os resultdos em percentual para diferentes valores atribuídos ao *heartbeat*. O cenário do experimento, configurado com um *heartbaet* de 15 segundos, foi o que apresentou maior influência no tempo de finalização da aplicação paralela, por realizar um monitoramento frequente dos processos envolvidos na computação.

	Nº processos			
heartbeat (s)	4	8	12	20
15	1,67	1,87	1,92	2,62
30	1,11	1,26	1,52	2,45
45	0,93	1,03	0,91	1,81
60	0,56	0,80	0,76	0,87

Tabela 5.2: Percentual de influência do componente DF. Fonte: Autor.

A partir da Figura 5.2, observa-se a influência no desempenho da aplicação paralela, quando configurada com o valor do *heartbeat* de 15 segundos, com um acréscimo de até 2,6% no tempo final da aplicação. Já para o cenário configurado com o valor do *heartbeat* de 60 segundos, pode-se observar que a influência no desempenho da aplicação paralela não ultrapassa 0,9% no tempo final da aplicação.

5.1.2 Tempo de detecção de falhas

Esse experimento visa avaliar a influência do componente DF em um cenário de execução com falhas. As falhas foram injetadas nos processos monitorados, forçando esses processos a não enviarem as mensagens de *heartbeat*, a partir de um determinado momento, aos processos monitores. Nesse experimento, o componente DF foi configurado com valores para o *heartbeat* iguais a 15, 30, 45 e 60 segundos, a aplicação, configurada com 4, 8, 12

e 20 processos. A Tabela 5.3 apresenta a configuração dos cenários utilizados para este experimento.

heartbeat	Nº processos	Hora da falha	Hora da detecção
15 segundos	4	03:19:50	03:20:07
	8	03:21:06	03:21:27
	12	03:25:47	03:26:15
	20	03:30:26	03:30:52
30 segundos	4	03:39:40	03:40:31
	8	03:55:33	03:56:21
	12	03:57:19	03:58:06
	20	03:57:25	03:58:06
45 segundos	4	04:39:46	04:41:03
	8	04:41:23	04:42:43
	12	04:43:02	04:44:23
	20	04:45:23	04:46:53
60 segundos	4	04:45:15	04:46:39
	8	04:47:32	04:48:59
	12	04:49:02	04:50:50
	20	04:53:15	04:54:59

Tabela 5.3: Cenários utilizados para o componente DF. Fonte: Autor

O cenário do experimento configurado com o valor do *heartbeat* de 15 segundos é que apresenta um melhor resultado na detecção de falhas, por realizar um monitoramento frequente dos processos envolvidos na computação. Note que um frequente monitoramento pode levar a um aumento no tempo de execução da aplicação e, consequentemente, levará mais tempo, para completar o processamento da aplicação.

Na Figura 5.5, observa-se que quanto maior o valor do *heartbeat*, maior o tempo gasto na detecção da falha. Isso porque, quanto maior o valor do *heartbeat* menor será a frequencia de monitoramento dos processos.

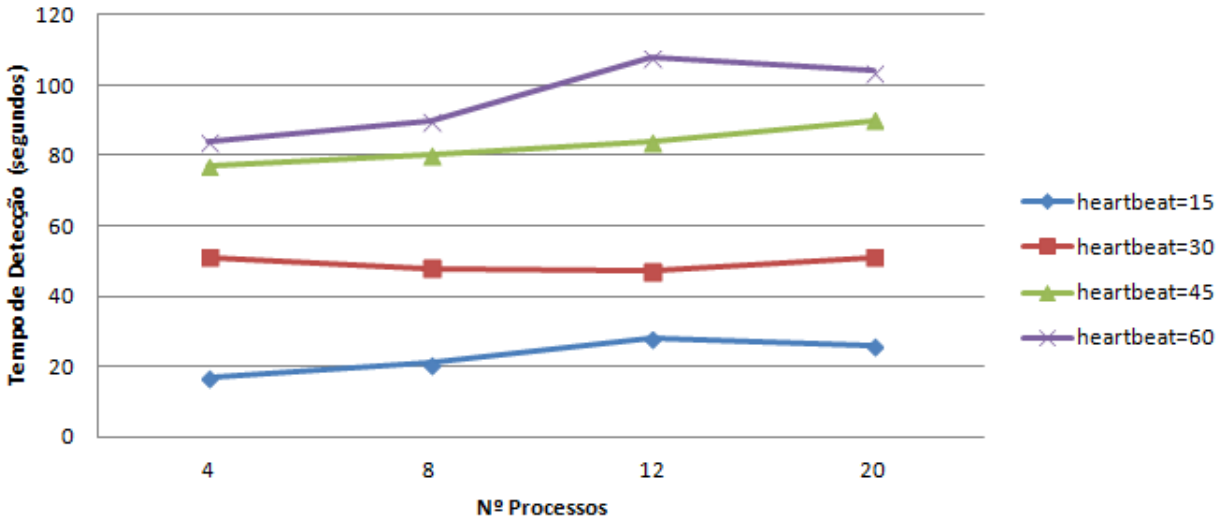

Figura 5.5: Tempo de detecção de falhas. Fonte: Autor.

5.2 Influência dos parâmetros do mecanismo de checkpoint

Para avaliar a influência do mecanismo de *checkpoint* do componente GD, foram realizados experimentos em um cenário livre de falhas, com diferentes valores de tempo para a tomada de *checkpoint*, variando o número de processos participativos na computação paralela distribuída.

5.2.1 Influência do componente GD em um cenário livre de falhas

Esse experimento visa avaliar a influência do componente GD em um cenário de execução livre de falhas. Nesse experimento, o componente GD foi configurado com valores para o *checkpointTime* iguais a 30, 60, 90 e 120 segundos, a aplicação foi configurada com 4, 8, 12 e 20 processos. A Figura 5.6 apresenta o tempo de execução da aplicação, apenas, com o componente GD ativo.

Cada coluna da Figura 5.6 representa o tempo total de execução da aplicação paralela em determinada situação. A coluna "SGD" representa a execução da aplicação sem o GD ativo, ou seja, trata-se de uma execução normal da aplicação original sem a injeção de qualquer componente do ACTF. A coluna "checkpointTime=30" representa a execução da aplicação paralela com o valor da variável *checkpointTime* do GD configurada para que, a cada 30 segundos, seja realizada a operação de tomada de *checkpoint* dos processos paralelos. O mesmo ocorre para as colunas "checkpointTime=60", "checkpointTime=90" e "checkpointTime=120", tendo, apenas, a periodicidade da tomada dos *checkpoints* vari-

ada em função dos valores do *checkpointTime*. Com os valores obtidos nesse experimento, pode-se observar que, quanto menor o valor do *checkpointTime*, maior é a influência do mecanismo de *checkpoint* no tempo final do processamento da aplicação paralela.

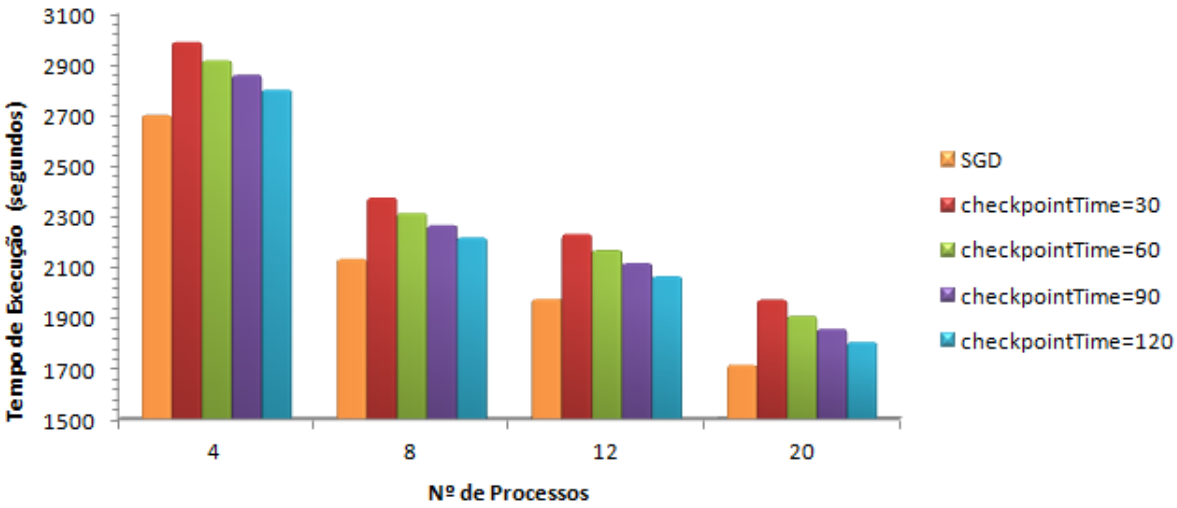

Figura 5.6: Tempo de execução em um cenário livre de falhas com o GD ativo. Fonte: Autor.

O cenário do experimento, configurado com um *checkpointTime* de 120 segundos, foi o que apresentou menor influência no tempo de finalização da aplicação paralela, por ter uma menor frequência na execução dos *checkpoints*. Na Figura 5.7, pode-se observar que, quanto maior o número de processos participativos na computação, maior é a influência do mecanismo no desempenho da aplicação paralela.

A Tabela 5.4 apresenta os resultdos em percentual para diferentes valores atribuídos à variável *checkpointTime*. Note que, quanto maior o valor do *checkpointTime*, menor é a influência do mecanismo de *checkpoint* no tempo final do processamento da aplicação paralela.

	Nº processos			
checkpointTime (s)	4	8	12	20
30	10,67	11,33	13,00	15,00
60	8,00	8,50	9,75	11,25
90	5,87	6,23	7,15	8,25
120	3,73	3,97	4,55	5,25

Tabela 5.4: Percentual de influência do componente GD. Fonte: Autor.

É possível observar, na Figura 5.7, que a influência no desempenho da aplicação paralela, quando configurada com o valor do *checkpointTime* de 30 segundos, ocorre um acréscimo de até 15% no tempo final da aplicação, por realizar uma frequente tomada de *checkpoints* dos processos paralelos. Já, para o cenário configurado com o valor do *checkpointTime* de 120 segundos, pode-se observar que a influência no desempenho da aplicação paralela

é de no máximo 5% no tempo final da aplicação.

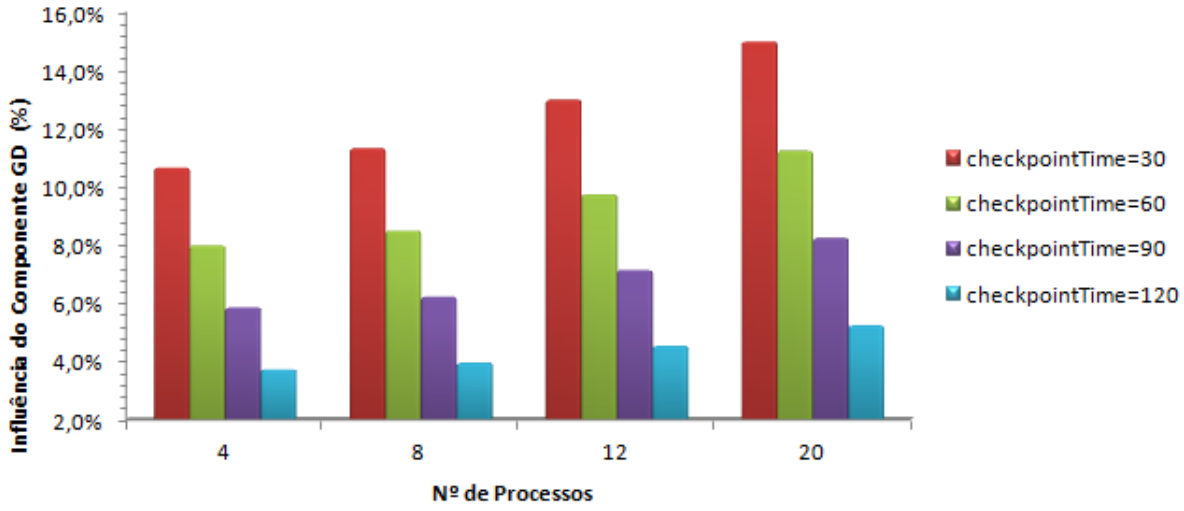

Figura 5.7: Influência do componente GD em um cenário livre de falhas. Fonte: Autor.

5.3 *Influência do ACTF na sobrecarga da aplicação paralela*

Esse experimento visa avaliar a sobrecarga na aplicação paralela, provocada pelo uso do mecanismo de tolerância a falhas, implementado no ACTF. Para tanto, comparou-se o tempo da aplicação paralela, sem a utilização do mecanismo do ACFT e com a utilização do mecanismo de tolerância a falhas do ACTF, ou seja, uma aplicação compilada, fazendo uso dos serviços do DF, GD e GP. O *heartbeat* foi configurado com o valor de 45 segundos, por se tratar de um valor intermediário entre o menor valor testado que foi de 15 segundos e o maior valor testado que foi de 60 segundos, conforme experimento apresentado na Seção 5.1.1.

O tempo para a tomada de *checkpoint* dos processos paralelos foi configurado com o valor de 90 segundos, por se tratar de um valor intermediário entre o menor valor testado que foi de 30 segundos e o maior valor testado que foi de 120 segundos, conforme experimento apresentado na Seção 5.2.1. Foram utilizados 20 processos em cada execução da aplicação paralela, por se tratar do número máximo de processos utilizados nos experimentos.

Os testes foram executados, variando-se o tamanho da imagem formando uma matriz (n x n). A Tabela 5.5 ilustra os valores de sobrecarga do mecanismo de tolerância a falhas para a aplicação paralela de processamento digital de imagem. A coluna “STF” representa os valores dos tempos para a aplicação paralela sem o mecanismo de tolerância a falhas. Já a coluna “CFT” representa os valores da aplicação paralela com a utilização do mecanismo de tolerância a falhas.

Número de pixel (n x n)	STF (s)	CTF (s)	Sobrecarga(%)
1000	520,00	534,60	3
2000	600,00	618,00	3
3000	734,00	763,36	4
4000	924,00	979,44	6
5000	1145,00	1248,15	8
6000	1462,00	1666,68	14

Tabela 5.5: Valores da sobrecarga do mecanismo de tolerância a falhas. Fonte: Autor

A partir dos dados coletados pelo experimento, observa-se, a partir da Figura 5.8, que, para as imagens com tamanho entre 1000 e 3000 (linhas x colunas), a sobrecarga imposta pelo mecanismo é menor que 4% no tempo final de execução da aplicação paralela. Já para o cenário onde tivemos uma imagem de 6000 (linhas x colunas), tivemos uma sobrecarga de 14% no tempo final de execução da aplicação paralela.

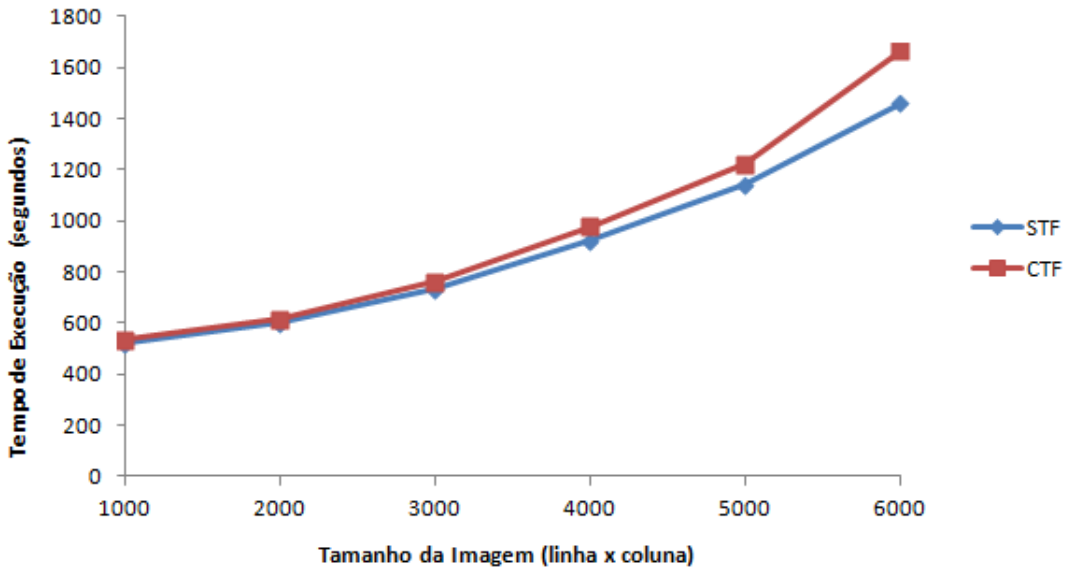

Figura 5.8: Sobrecarga do mecanismo de *checkpoint* para diferentes tamanhos de imagem. Fonte: Autor.

O mecanismo de tolerância a falhas do ACTF impõe sobrecarga na aplicação paralela distribuída. Note que temos a inclusão de dois componentes importantes, para garantir a continuidade da aplicação na ocorrência de falhas: tanto o DF quanto o GD contribuem com o tempo extra, para a finalização da computação paralela.

O componente GD é o que mais influência na sobrecarga da aplicação paralela. Quando o GD solicita a tomada de um *checkpoint*, os processos paralelos que compõem a aplicação devem sofrer uma pausa e somente retornarem ao processo, quando a operação de gravação dos contextos dos processos for concluída. Essa parada temporária reflete-se diretamente no tempo final necessário para a conclusão do processamento. Por esta razão, pode-se concluir que uma aplicação paralela, que venha a utilizar o ambiente ACTF, sofrerá um

atraso no tempo de conclusão de seu processamento, sendo este atraso determinado pela quantidade e pela periodicidade dos *checkpoints* tomados.

5.4 Tempo de reconfiguração da aplicação paralela

Esse experimento visa avaliar o tempo de reconfiguração da aplicação paralela em um cenário de execução com falhas. Nesse experimento, o componente DF foi configurado com o valor para o *heartbeat* de 60 segundos, por se tratar do melhor valor no experimento apresentado na Seção 5.1.1. O componente GD foi configurado com o valor para o *checkpointTime* igual a 120 segundos, por se tratar do melhor valor encontrado no experimento apresentado na Seção 5.2.1. As falhas foram injetadas 10 minutos após a inicialização dos processos paralelos, para garantir que todos os processos estivessem em pleno uso dos recursos computacionais. A aplicação foi configurada com 20 processos, sendo 4 processos distribuídos para cada nó do *cluster* de computadores. O tempo de reconfiguração da aplicação paralela foi calculado, obedecendo-se aos seguintes critérios, após a detecção de uma falha:

1. Propagação da falha;
2. Reconstrução do arquivo de *status* das maquinas do *cluster*;
3. Solicitação de parada dos processos paralelos;
4. Solicitação de recuperação de todos os processos da aplicação;
5. Inicialização da aplicação paralela em uma nova configuração do *cluster*.

A partir da Figura 5.9, é possível observar o tempo de reconfiguração da aplicação paralela, após a ocorrência de falhas. A coluna "PF" representa o tempo que o DF leva para propagar a falha para todos os nós participativos da computação paralela; a coluna "RA" representa o tempo para a reconstrução do arquivo de *hosts* de cada máquina do *cluster* de computadores - a reconstrução do arquivo se baseia na exclusão do nó falho; a coluna "SP" representa o tempo da solicitação de parada de todos os processos paralelos, que permaneceram ativos após a detecção da falha, ou seja, todos os processos, com exceção dos processos paralelos, que estavam no nó falho.

A coluna "SR" representa o tempo de solicitação de recuperação dos estados de todos os processos da aplicação paralela trata-se da transferência dos arquivos de contexto armazenados do SATF; a coluna "IA" representa o tempo que a aplicação paralela leva para inicializar todos os processos em um novo conjunto de máquinas.

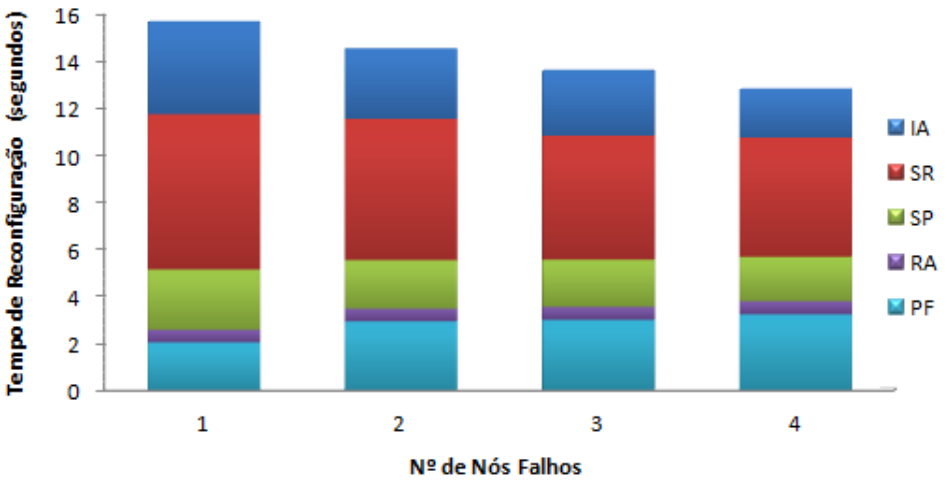

Figura 5.9: Tempo de reconfiguração da aplicação paralela em um cenário com falhas. Fonte: Autor.

A Tabela 5.6 apresenta o tempo de reconfiguração da aplicação paralela por categoria, podemos notar que para a categoria "IA" que representa o tempo que a aplicação paralela leva para inicializar todos os processos em um novo conjunto de máquinas foi de no máximo 3,99 segundos.

Categoria	Nº de nós faltosos			
	1	2	3	4
IA	3,99	3,03	2,81	2,10
SR	6,60	6,01	5,28	5,08
SP	2,55	2,05	2,01	1,89
RA	0,55	0,55	0,55	0,55
PF	2,03	2,93	2,99	3,23
Tempo total	15,72	14,57	13,64	12,85

Tabela 5.6: Tempo de reconfiguração da aplicação paralela. Fonte: Autor.

A categoria "SR" que representa o tempo de solicitação de recuperação dos estados de todos os processos da aplicação paralela, obteve o tempo máximo de 6,60 segundos, por se tratar da transferência dos arquivos de contexto armazenados do SATF para os nós remanescentes do *cluster* de computadores. Já o tempo de solicitação de parada de todos os processos paralelos "SP", se manteve entre 1,89 segundos quando tivemos quatro nós faltosos e 2,55 segundos quando tivemos a falta de um único nó. Esse fato ocorre, pois, com a falta de apenas um nó, configura uma quantidade maior de processos a serem notificados, para finalizarem suas atividades.

É possível observar que para a categoria "RA", a qual representa o tempo gasto para a reconstrução do arquivo de *hosts* de cada máquina do *cluster* de computadores se manteve constante em 0,55 segundos, independentemente do número de nós faltosos. Já o tempo que o DF leva para propagar a falha para todos os nós participativos da computação

paralela, representado pela categoria “PF”, se manteve entre 2,03 segundos para a falta de um nó e 3,23 segundos quando tivemos a falta de quatro nós.

Capítulo Seis

Considerações finais

6.1 Conclusões

O ambiente proposto nesta pesquisa, através dos experimentos realizados, como detalhado no Capítulo 5, mostrou ser possível tratar falhas da classe *fail stop*, sem acrescentar ou propor qualquer mudança no padrão Open MPI, fazendo uso da técnica de *checkpoint/restart* fornecida pelo próprio padrão Open MPI. A partir da utilização do ACTF, é possível suprir a falta de conhecimento sobre os mecanismos de tolerância a falhas dos usuários programadores de aplicações paralelas que utilizam o padrão Open MPI. Outras pesquisas, conforme apresentadas no Capítulo 3 se propuseram a implementar técnicas de tolerância a falhas no MPI, tais como *checkpoint/restart*, *log* de mensagens e/ou através de modificações na semântica do padrão. A estratégia de detecção de falhas implementada no ACTF foi inspirada no mecanismo proposto pela arquitetura RADIC (DUARTE; REXACHS; LUQUE, 2006), por se tratar de um mecanismo escalável.

Conforme os objetivos propostos inicialmente nesta pesquisa, foram desenvolvidos dois subsistemas, um subsistema de detecção de falhas e outro para automatizar os procedimentos de *checkpoint/restart* do Open MPI definidos por Hursey (HURSEY, 2010). O componente GD foi desenvolvido para automatizar os pedidos de tomada de *checkpoints* dos processos paralelos. Pôde-se notar conforme detalhado na Seção 5.2, que o componente GD apresentou um *overhead* de 15% no tempo final da aplicação paralela quando o ACTF foi configurado com o valor do *checkpointTime* igual a 30 segundos. Porém, quando o ACTF foi configurado com o valor do *checkpointTime* igual a 120 segundos, o *overhead* foi de apenas 5%.

O componente GP foi desenvolvido para executar o procedimento de *restart* da aplicação paralela, em caso de falha. O DF é o componente responsável pela detecção de falhas da classe *fail stop*, permitindo que os nós participantes do *cluster* de computadores possam ser monitorados. Esse componente gerou um *overhead* de até 2,6% no tempo final da aplicação paralela, quando configurada com o valor do *heartbeat* igual a 15 segundos. Porém, quando o ACTF foi configurado com o valor do *heartbeat* igual a 60 segundos, pôde-se observar que o *overhead* não ultrapassou 0,9%, conforme detalhado na Seção 5.1.

Foi detectado que o ACTF acrescenta uma sobrecarga no tempo de conclusão do processamento da aplicação paralela, em decorrência do tempo gasto para o monitoramento do ambiente de execução dos processos paralelos e para as tomadas de *checkpoints*. Porém essa sobrecarga pode ser reduzida, aumentando-se as periodicidades de monitoramento

dos processos e de tomada dos *checkpoints* desses mesmos processos. No caso dos experimentos realizados, o aumento no tempo de execução da aplicação paralela, para um cenário livre de falhas, se manteve entre 4% e 16%, mas o percentual de 16% atingiu tal valor, em decorrência das tomadas dos *checkpoints*, que foram muito frequentes.

6.2 Contribuições

Os objetivos planejados inicialmente foram atingidos, ou seja, a construção de um ambiente computacional tolerante a falhas, para aplicações paralelas que utilizam o padrão Open MPI, fornecendo mecanismos para minimizar o impacto sobre o desempenho das aplicações paralelas distribuídas, evitando degradação, quando as falhas ocorrerem ao longo da execução da aplicação paralela, sendo esta a principal contribuição deste trabalho.

O ACTF disponibiliza aos usuários programadores de computação de alto desempenho que utilizam o padrão Open MPI, um ambiente de execução que permite a geração automática de pontos de recuperação (*checkpoints*), em intervalos de tempo pré-definidos pelo usuário, o monitoramento dos nós participativos da computação paralela, detecção de falhas da classe *fail stop* e a reativação automática dos processos paralelos distribuídos, preservando-se o processamento realizado até o último *checkpoint* armazenado no sistema de arquivos tolerantes a falhas, em caso de falhas.

A pesquisa apresenta uma análise sobre as implementações de sistemas de tolerância a falhas em bibliotecas MPI e uso justificado da implementação mais adequada para tolerância a falhas.

Como contribuição secundária, tem-se a disponibilização de uma interface gráfica, que permite, ao usuário programador, compilar e executar o seu programa paralelo, incluindo os componentes necessários, para prover tolerância a falhas, sem a necessidade de conhecer qualquer comando específico de inicialização dos processos paralelos baseados no padrão Open MPI.

6.3 Atividades Futuras de Pesquisa

Como proposta de atividade futura desta pesquisa, uma questão importante a ser tratada é a de que o próprio ambiente possa calcular o melhor momento para a tomada dos *checkpoints*, baseado no comportamento, carga e utilização da memória imposta pela própria aplicação paralela. Quanto menor a ocupação da memória, menor são os arquivos de con-

textos e, consequentemente, menor a utilização da rede no momento de armazenamento dos estados dos processos paralelos.

O ambiente computacional desenvolvido durante essa pesquisa trata apenas de falhas da classe *fail stop*. Uma possível proposta de continuidade desse trabalho seria a possibilidade de o ambiente suportar outras classes de falhas.

Outra proposta de atividade futura, seria a implementação de um algoritimo alternativo para a coordenação dos *checkpoints*. Algoritmos de coordenação alternativos de *checkpoint* muitas vezes permitem uma solução mais escalável de *checkpoint/restart* (GAO et al., 2007; VAIDYA, 1999).

Uma última sugestão, como atividade futura dessa pesquisa, seria a possibilidade de testar o ACTF com outros modelos de aplicação, principalmente com comunicações heterogêneas e não bloqueantes.

Apêndice A

Modelo de Análise do ACTF

A.1 Diagrama de caso de uso

A Figura A.1 ilustra os atores e casos de uso do ACTF. O ator "Programador"representa o usuário programado, que utilizará o caso de uso "Modificar Estado da Aplicação Paralela", para incluir os componentes necessários a tornar a aplicação paralela escrita no padrão Open MPI, tolerante a falhas. O ator "Sistema Operacional"é responsável por escalonar, periodicamente, os serviços de detecção de falhas e tomadas dos checkpoints, representados, respectivamente, pelos casos de uso "Detectar Falhas"e "Realizar Checkpoint". O ator "Mpirun"representa a interface do Open MPI utilizada pelo ACTF, para realizar o restart da aplicação paralela, em caso de falha - o caso de uso "Restaurar Aplicação Paralela"representa esse requisito.

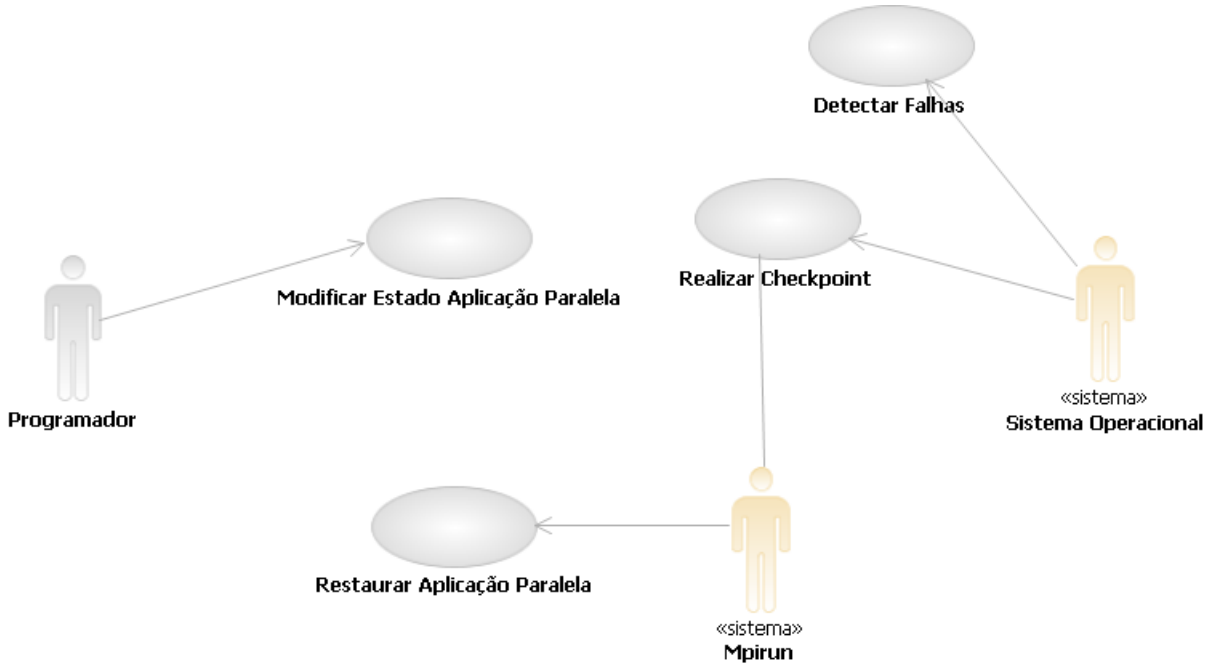

Figura A.1: Diagrama de caso de uso. Fonte: Autor.

A.2 Realização do caso de uso Detectar Falhas

A Figura A.2 ilustra as classes cujos objetos participam da realização do caso de uso Detectar Falhas.

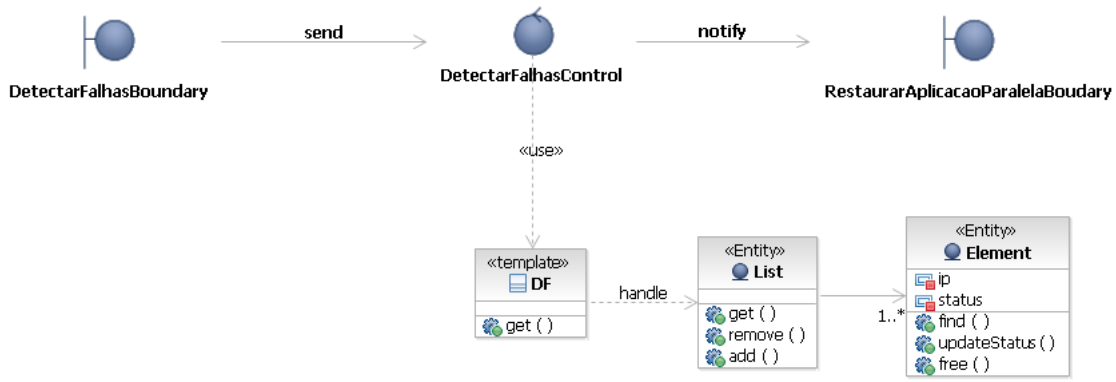

Figura A.2: Diagrama de classes - Detectar Falhas. Fonte: Autor.

A Figura A.3 ilustra os objetos participando em interações de acordo com suas linhas de vida e as mensagens que trocam para a realização do fluxo **Enviar Mensagem de Heartbeat** do caso de uso Detectar Falhas.

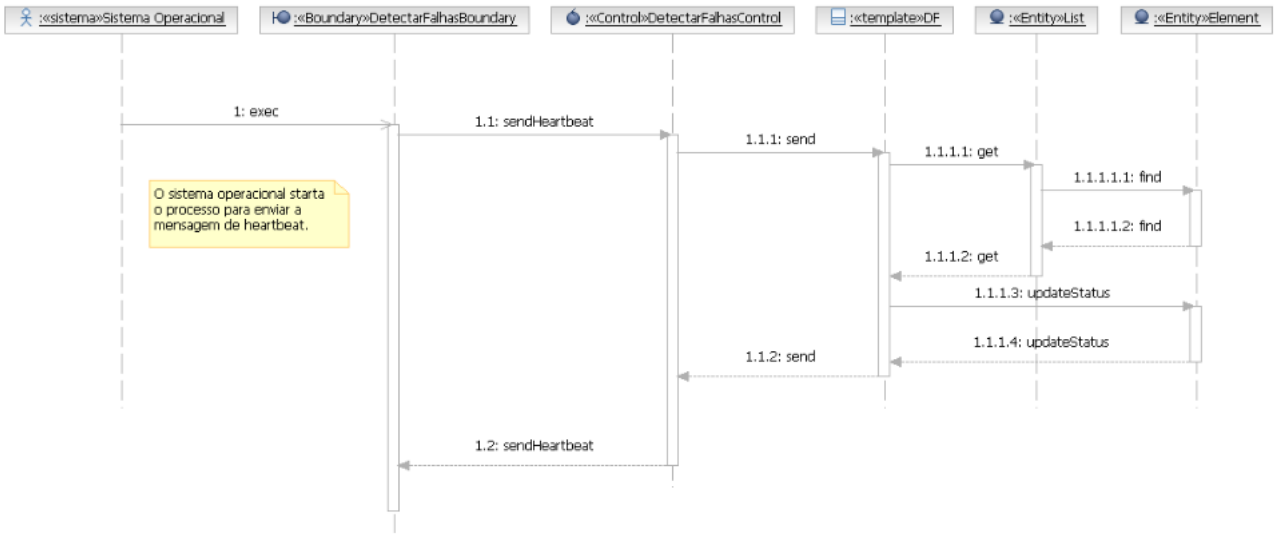

Figura A.3: Diagrama de sequência - Enviar Mensagem de Heartbeat. Fonte: Autor.

A Figura A.4 ilustra os objetos participando em interações de acordo com suas linhas de vida e as mensagens que trocam para a realização do fluxo **Detecção da Falha** do caso de uso Detectar Falhas.

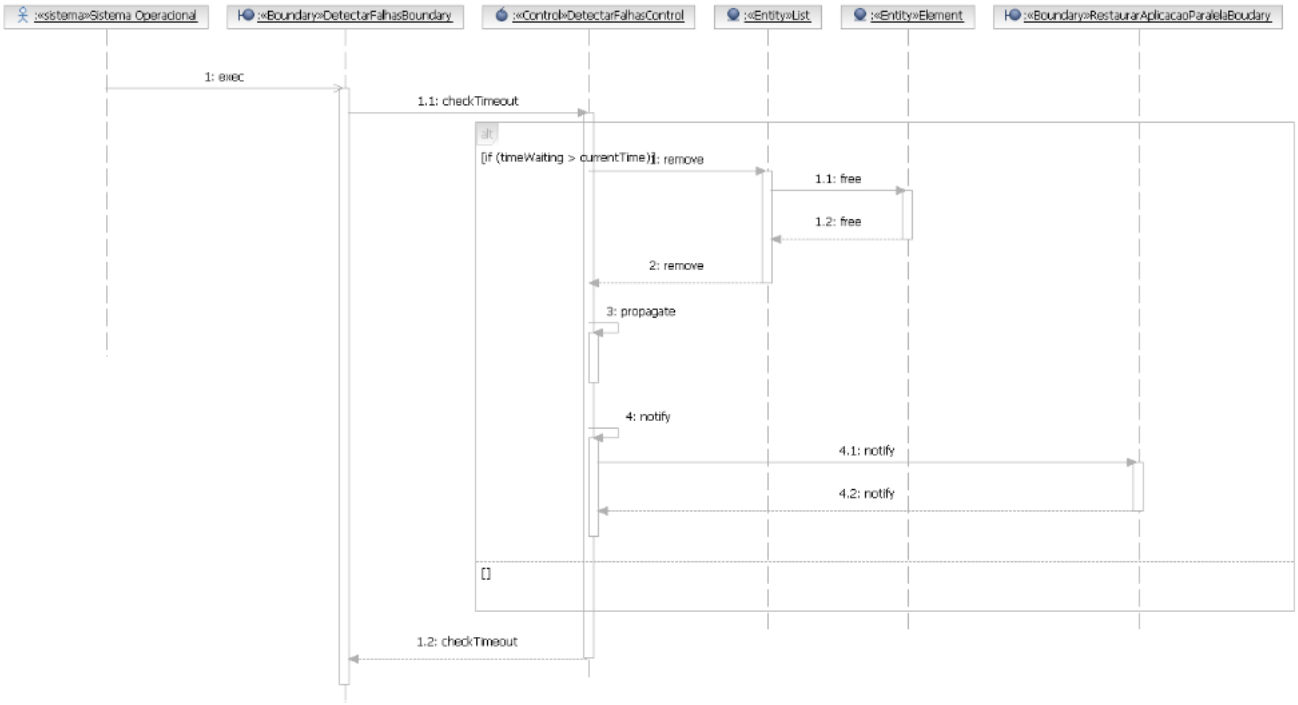

Figura A.4: Diagrama de sequência - Detecção da Falha. Fonte: Autor.

A.3 Realização do caso de uso Modificar Estado da Aplicação Paralela

A Figura A.5 ilustra as classes cujos objetos participam da realização do caso de uso Modificar Estado da Aplicação Paralela.

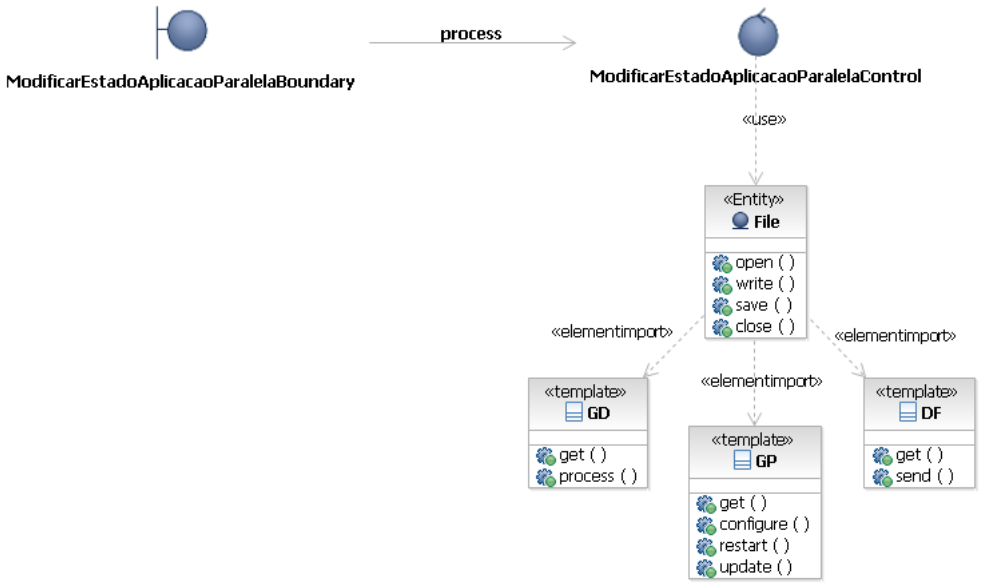

Figura A.5: Diagrama de classes participantes - Modificar Estado da Aplicação Paralela. Fonte: Autor.

A Figura A.6 ilustra os objetos participando em interações de acordo com suas linhas de vida e as mensagens que trocam para a realização do fluxo **Transformar Aplicação** do caso de uso Modificar Estado da Aplicação Paralela.

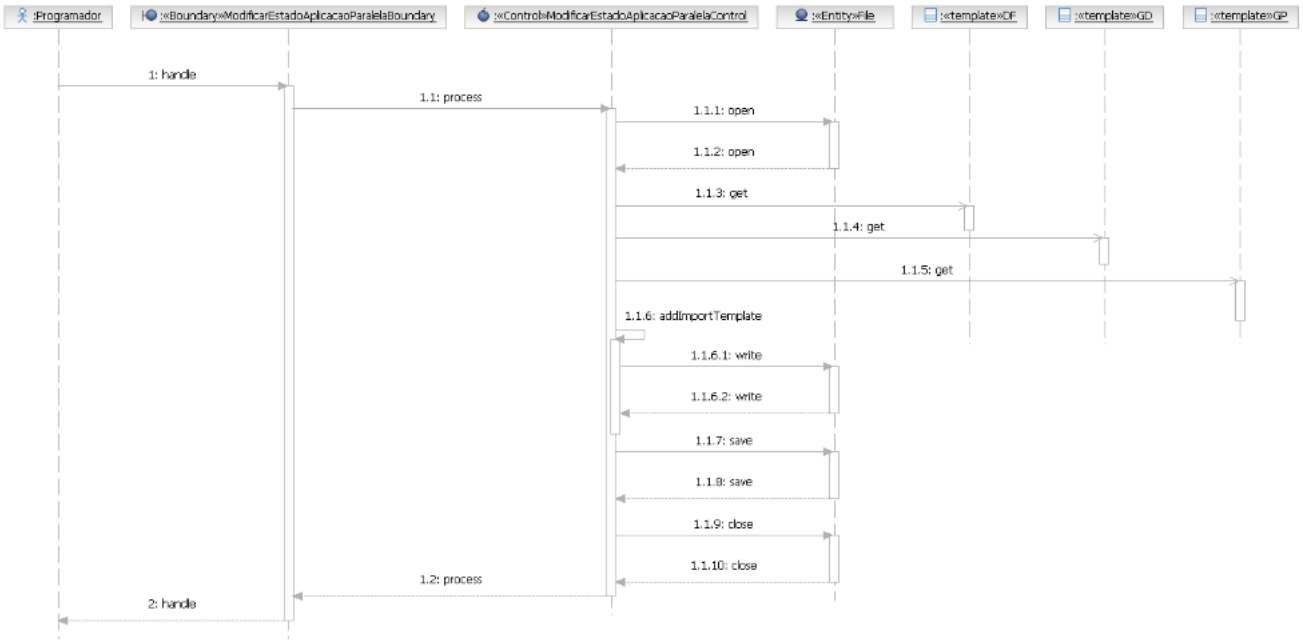

Figura A.6: Diagrama de seqüência - Transformar Aplicação. Fonte: Autor.

A.4 Realização do caso de uso Realizar Checkpoint

A Figura A.7 ilustra as classes cujos objetos participam da realização do caso de uso Realizar Checkpoint.

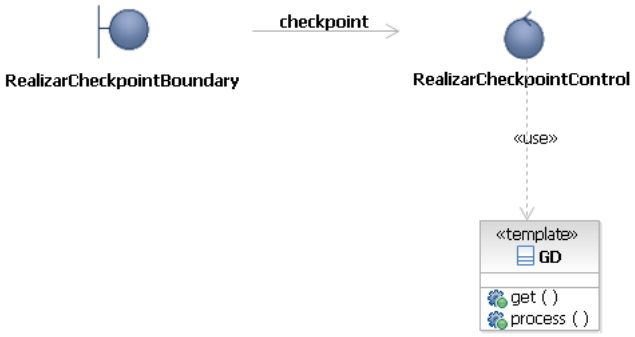

Figura A.7: Diagrama de classes - Realizar Checkpoint. Fonte: Autor.

A Figura A.8 ilustra os objetos participando em interações de acordo com suas linhas de vida e as mensagens que trocam para a realização do fluxo **Realizar Checkpoint** do caso de uso Realizar Checkpoint.

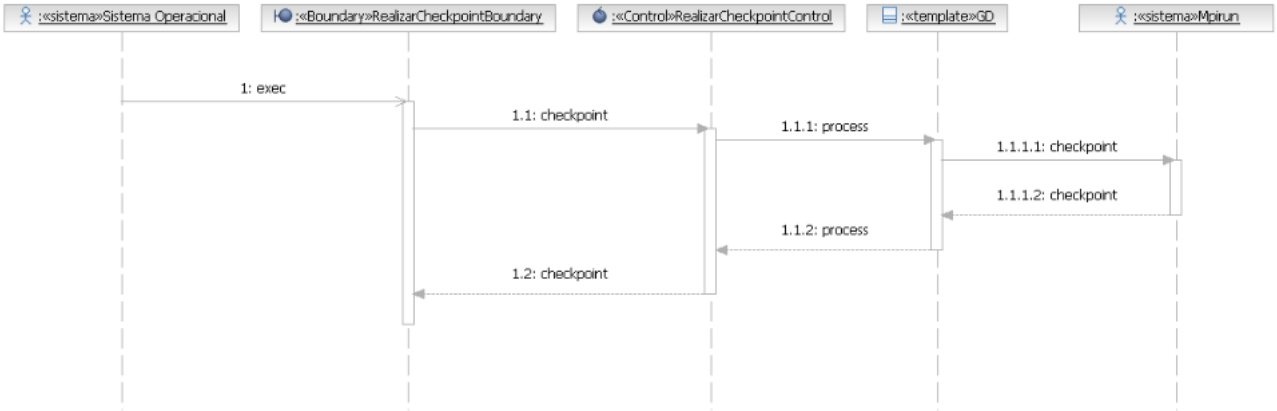

Figura A.8: Diagrama de seqüência - Realizar Checkpoint. Fonte: Autor.

A.5 Realização do caso de uso Restaurar Aplicação Paralela

A Figura A.9 ilustra as classes cujos objetos participam da realização do caso de uso Restaurar Aplicação Paralela.

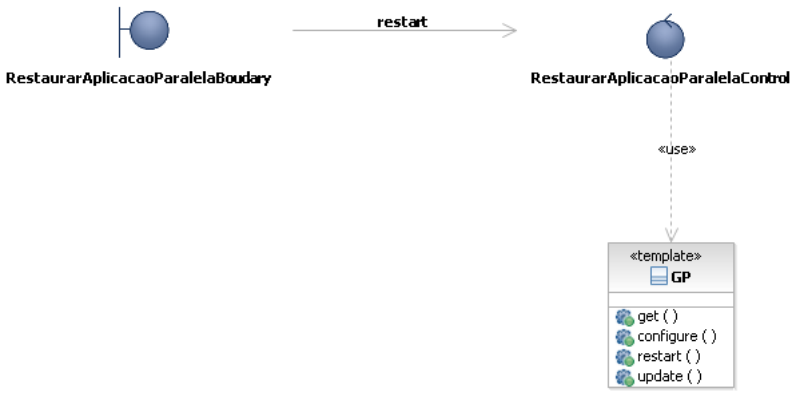

Figura A.9: Diagrama de classes - Restaurar Aplicação Paralela. Fonte: Autor.

A Figura A.10 ilustra os objetos participando em interações de acordo com suas linhas de vida e as mensagens que trocam para a realização do fluxo **Restart Aplicação** do caso de uso Restaurar Aplicação.

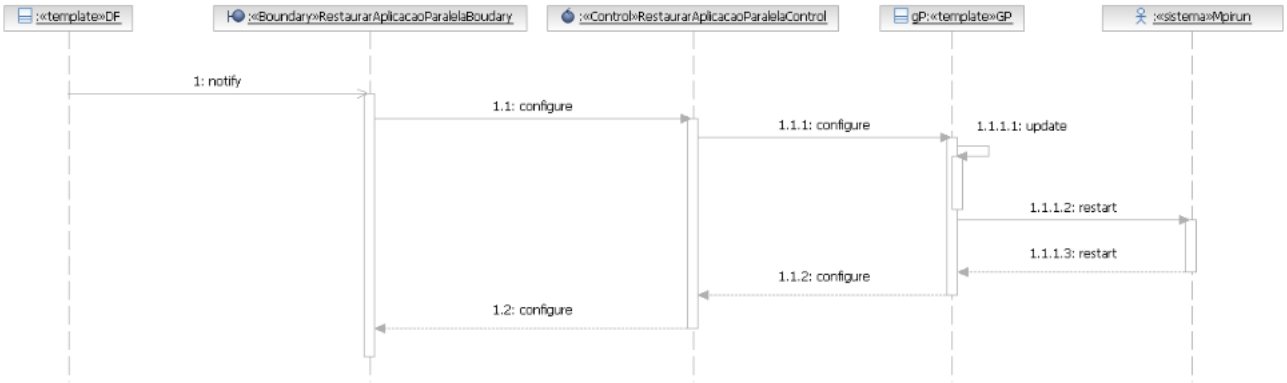

Figura A.10: Diagrama de seqüência - Restart Aplicação. Fonte: Autor.

Anexo B

OMPI-CHECKPOINT

O comando **ompi-checkpoint** é utilizado para a realização de *checkpoints* de uma aplicação escrita no padrão MPI. O argumento exigido para este comando é o PID, do processo gerado através do comando **mpirun**. Uma vez que um pedido de *checkpoint* completou sua execução, o ompi-checkpoint retornará uma referência instantânea global. Esta informação permitirá que os processos sejam reiniciados corretamente em um momento posterior.

A Figura B.1 apresenta a interface de comando para a tomada de *checkpoint*, através do comando **ompi-checkpoint**.

```
ompi-checkpoint PID_OF_MPIRUN \
    [-h | --help]
    [-v | --verbose]
    [-V #]
    [--term]
    [--stop]
    [-w | --nowait]
    [-s | --status]
    [-l | --list]
    [-attach | --attach]
    [-detach | --detach]
    [-crdebug | --crdebug]
```

Figura B.1: Comando ompi-checkpoint. Fonte: OPENMPI (2012).

A Figura B.2 apresenta um breve exemplo de como o comando **ompi-checkpoint** poderia ser usado para uma aplicação MPI. O comando gera dois pontos de controle, entre 0 e 1, e uma referência global, **ompi-global-snapshot-1234**.

```
shell$ mpirun my-app <args> &
shell$ export PID_OF_MPIRUN=1234
shell$ ompi-checkpoint $PID_OF_MPIRUN
Snapshot Ref.: 0 ompi-global-snapshot-1234
shell$ ompi-checkpoint $PID_OF_MPIRUN
Snapshot Ref.: 1 ompi-global-snapshot-1234
```

Figura B.2: Exemplo do comando ompi-checkpoint. Fonte: OPENMPI (2012).

Note que o comando **ompi-checkpoint** recebe alguns argumentos de entrada, para completar sua execução. A Tabela B.1 apresenta a definição de cada parâmetro.

Argumento		Descrição
PID_OF_MPIRUN		PID do processo mpirun.
-h	–help	Exibir ajuda.
-v	–verbose	Exibir os detalhes da saída.
-V #		Exibir os detalhes da saída até um nível especificado.
–term		Encerrar o aplicativo depois do *checkpoint*.
–stop		Enviar SIGSTOP a aplicação logo após o ponto de verificação (*checkpoint* não vai terminar até SIGCONT ser enviado) (Não pode ser usado com – term)
-w	–nowait	Não Implementado.
-s	–status	Exibir mensagens de *status* que descreve a progressão do ponto de verificação.
-l	–list	Exibir uma lista de arquivos de ponto de verificação disponíveis nesta máquina.
-attach	–attach	Aguarda o depurador anexar diretamente depois de tomar o ponto de verificação. **Incluído no v1.5.1 e versões posteriores**.
-detach	–detach	Não espere para o depurador para recolocar depois de tomar o ponto de verificação. **Incluído no v1.5.1 e versões posteriores**.
-crdebug	–crdebug	Ativar C/R Depuração Avançada.

Tabela B.1: Argumentos do comando ompi-checkpoint. Fonte: OPENMPI (2012).

Anexo C

OMPI-RESTART

O comando **ompi-restart** é fornecido para reiniciar a aplicação, baseado em um *checkpoint* da aplicação MPI. O argumento exigido para este comando é a referência global, retornado pelo comando **ompi-checkpoint**. A referência global contém todas as informações necessárias para reiniciar uma aplicação MPI. A invocação do comando **ompi-restart** resultará em um novo **mpirun** a ser lançado.

A Figura C.1 apresenta a interface de comando para a tomada de *restart*, através do comando **ompi-restart**.

```
ompi-restart GLOBAL_SNAPSHOT_REF \
    [-h | --help]
    [-v | --verbose]
    [--fork]
    [-s | --seq]
    [--hostfile]
    [--machinefile]
    [-i | --info]
    [-a | --apponly]
    [-crdebug | --crdebug]
    [-mpirun_opts | --mpirun_opts]
    [--showme]
```

Figura C.1: Comando ompi-restart. Fonte: OPENMPI (2012).

A Figura C.2 apresenta um breve exemplo de como o comando **ompi-restart** poderia ser usado para uma aplicação MPI. O comando solicita o *restart* de estado global, definido como **ompi-global-snapshot-1234**.

```
shell$ ompi-restart ompi-global-snapshot-1234
```

Figura C.2: Exemplo do comando ompi-restart. Fonte: OPENMPI (2012).

Note que o comando **ompi-restart** recebe alguns argumentos de entrada, para completar sua execução. A Tabela C.1 apresenta a definição de cada parâmetro.

Argumento		Descrição
GLOBAL_SNAPSHOT_REF		Referência global instantânea
-h	–help	Exibir ajuda.
-v	–verbose	Exibir os detalhes da saída.
–fork		Garfo fora um novo processo que é reiniciado em vez de substituir orte_restart.
-s	–seq	O número de seqüência do ponto de verificação para começar. (Padrão: -1, ou mais recente).
–hostfile	–machinefile	Fornecer um *hostfile* usar para o lançamento.
-i	–info	Exibir informações sobre o ponto de verificação.
-a	-apponly	Só criar o arquivo de contexto app, não reinicie a partir dele. **Incluído no v1.5.1 e versões posteriores**.
-crdebug	–crdebug	Ativar C/R Depuração aprimorada. **Incluído no v1.5.1 e versões posteriores**.
-mpirun_opts	–mpirun_opts	Opções de linha de comando para passar diretamente para mpirun. **Incluído no v1.5.1 e versões posteriores**.
- Showme		Mostrar a linha de comando completa que teria sido executada.

Tabela C.1: Argumentos do comando ompi-restart. Fonte: OPENMPI (2012).

Referências Bibliográficas

ADNAN, A.; ROY, F. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. *Cluster Computing*, v. 6, p. 227–236, 2003.

AULWES, R. et al. Architecture of la-mpi, a network-fault-tolerant mpi. In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. [S.l.: s.n.], 2004. p. 15.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, v. 1, n. 22, p. 11–33, 2004. ISSN 1545-5971.

BATCHU, R.; DANDASS, Y. S.; SKJELLUM, A.; BEDDHU, M. Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, Kluwer Academic Publishers, Hingham, MA, USA, v. 7, n. 4, p. 303–315, out. 2004. ISSN 1386-7857. Disponível em: <http://dx.doi.org/10.1023/B:CLUS.0000039491.64560.8a>.

BOUTEILLER, A.; HERAULT, T.; KRAWEZIK, G.; LEMARINIER, P.; CAPPELLO, F. Mpich-v project: a multiprotocol automatic fault tolerant mpi. *International Journal of High Performance Computing Applications*, v. 20, n. 3, p. 319–333, 2006. Disponível em: <http://hal.inria.fr/hal-00688637>.

BRONEVETSKY, G.; MARQUES, D.; PINGALI, K.; STODGHILL, P. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, v. 38, n. 10, p. 84–94, 2003. ISSN 0362-1340.

CAPPELLO, F. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 23, n. 3, p. 212–226, ago. 2009. ISSN 1094-3420. Disponível em: <http://dx.doi.org/10.1177/1094342009106189>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, ACM, New York, NY, USA, v. 43, n. 2, p. 225–267, mar. 1996. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/226643.226647>.

CHANDY, K. M.; LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 3, n. 1, p. 63–75, fev. 1985. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/214451.214456>.

CHEN, W.; TOUEG, S.; AGUILERA, M. K. On the quality of service of failure detectors. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 51,

n. 5, p. 561–580, maio 2002. ISSN 0018-9340. Disponível em: <http://dx.doi.org/10-.1109/TC.2002.1004595>.

CRISTIAN, F. Understanding fault-tolerant distributed systems. *Commun. ACM*, ACM, New York, NY, USA, v. 34, n. 2, p. 56–78, fev. 1991. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/102792.102801>.

CRóSTA, A. P. *Processamento digital de imagens de sensoriamento remoto*. Campinas, SP: UNICAMP/Instituto de Geociências, 1993.

DUARTE, A.; REXACHS, D.; LUQUE, E. An intelligent management of fault tolerance in cluster using radicmpi. In: *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*. Berlin, Heidelberg: Springer-Verlag, 2006. (EuroPVM/MPI'06), p. 150–157. ISBN 3-540-39110-X, 978-3-540-39110-4. Disponível em: <http://dx.doi.org/10.1007/11846802_26>.

ELNOZAHY, E. N. M.; ALVISI, L.; WANG, Y.-M.; JOHNSON, D. B. A survey of rollbackrecovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, v. 24, n. 33, p. 375–408, 2002.

FAGG, G. E.; DONGARRA, J. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In: *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2000. p. 346–353. ISBN 3-540-41010-4. Disponível em: <http://dl.acm.org/citation.cfm?id=648137.746632>.

GAO, Q.; HUANG, W.; KOOP, M. J.; PANDA, D. K. Group-based coordinated checkpointing for mpi: A case study on infiniband. In: *Parallel Processing, 2007. ICPP 2007. International Conference on*. [S.l.: s.n.], 2007.

GONZALEZ R., W. R. *Processamento de imagens digitais*. São Paulo: Edgard Blucher, 2000.

GREVE, F. G. P. *Protocolos Fundamentais para o Desenvolvimento de Aplicações Robustas*. Março 2005. URL: www.sbrc2007.ufpa.br/anais/2005/MC/cap7.pdf.

GäRTNER, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, v. 31, n. 26, p. 1–26, 1999.

GUPTA, I.; CHANDRA, T. D.; GOLDSZMIDT, G. S. On scalable and efficient distributed failure detectors. In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2001. (PODC '01), p. 170–179. ISBN 1-58113-383-9. Disponível em: <http://doi.acm.org/10.1145/383962-.384010>.

HARGROVE, P. H.; DUELL, J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, v. 46, p. 494–499, 2006.

HURSEY, J. *COORDINATED CHECKPOINT/RESTART PROCESS FAULT TOLERANCE FOR MPI APPLICATIONS ON HPC SYSTEMS.* Tese (Doctor of Philosophy) — Indiana University, Bloomington, 2010.

HURSEY, J.; SQUYRES, J.; MATTOX, T.; LUMSDAINE, A. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In: . [S.l.: s.n.], 2007. p. 1–8.

JOHNSON, D. B. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Tese (Doctor of Philosophy) — Rice University, Houston, 1989.

JULIANO, C. V.; WEBER, T. S. Injeção de falhas na fase de teste de aplicações distribuídas. *XX Simpósio Brasileiro de Engenharia de Software*, p. 161–176, 2006.

LOUCA, S.; NEOPHYTOU, N.; LACHANAS, A.; EVRIPIDOU, P. Mpi-ft: Portable fault tolerance scheme for mpi. *Parallel Processing Letters*, v. 10, n. 4, p. 371–382, 2000. Disponível em: <http://dblp.uni-trier.de/db/journals/ppl/ppl10.html>.

LUMPP J.E., J. Checkpointing with multicast communication. In: *Aerospace Conference, 1998 IEEE.* [S.l.: s.n.], 1998. v. 4, n. 12, p. 467–479. ISSN 1095-323X.

MAIER, J. Fault tolerance lessons applied to parallel computing. In: *Compcon Spring '93, Digest of Papers.* [S.l.: s.n.], 1993. p. 244 –252.

MPI, F. *Message passing interface (MPI) Forum.* 1994. URL: www.mpi-forum.org.

OPENMPI, O. M. *Open Source High Performance Computing.* Março 2012. URL: www.open-mpi.org.

PARK, T.; YEOM, H. Y. Application controlled checkpointing coordination for fault-tolerant distributed computing systems. *Parallel Computing*, v. 26, n. 15, p. 467–482, 2000.

PRADHAN, D. K. *Fault-tolerant computer system design.* New Jersey: Prentice-Hall, 1996.

SHIPMAN, G.; GRAHAM, R.; BOSILCA, G. Network fault tolerance in open mpi. In: KERMARREC, A.-M.; BOUGé, L.; PRIOL, T. (Ed.). *Euro-Par 2007 Parallel Processing.* Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4641). p. 868–878. ISBN 978-3-540-74465-8. Disponível em: <http://dx.doi.org/10-.1007/978-3-540-74466-5_93>.

SOUZA, J. R. *FTDR: Tolerancia a fallos, en clusters de computadores geográficamente distribuidos, basada en Replicación de Datos.* Tese (Doctor en Informática) — Universida Autònoma de Barcelona, Bellaterra, 2006.

SQUYRES, J. M. A component architecture for lam/mpi. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 38, n. 10, p. 2–, jun. 2003. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/966049.781510>.

SRIRAM, S.; JEFFREY, M. S.; BRIAN, B.; VISHAL, S.; ANDREW, L. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, v. 19, p. 479–493, 2005.

STELLING, P.; FOSTER, I.; KESSELMAN, C.; LEE, C.; LASZEWSKI, G. von. A fault detection service for wide area distributed computations. In: *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on.* [S.l.: s.n.], 1998. p. 268 –278.

TURNER, M. G. Landscape ecology: The effect of pattern on process. *Annual Review of Ecology and Systematics*, v. 20, n. 1, p. 171–197, 1989. Disponível em: <http://www.annualreviews.org/doi/abs/10.1146/annurev.es.20.110189.001131>.

VAIDYA, N. Staggered consistent checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, v. 10, n. 7, p. 694 –702, jul 1999. ISSN 1045-9219.

WANG, Y.-M.; HUANG, Y.; VO, K.-P.; CHUNG, P.-Y.; KINTALA, C. Checkpointing and its applications. In: *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1995. (FTCS '95), p. 22–. Disponível em: <http://dl.acm.org/citation.cfm?id=874064.875647>.

WEBER, R. F.; WEBER, T. S. Um experimento prático em programação diversitária. *III Simpósio em Sistemas de Computadores Tolerantes a Falhas (SCTF)*, p. 271–290, 1989.

Publicação

Fórum de Pós-Graduação

- PINHEIRO, O.; SOUZA, J. Um modelo computacional tolerante a falhas para aplicações paralelas utilizando mpi. In: ERAD-NE 2011 - FPG (). [s.n.], 2011. Disponível em: http://www.infojr.com.br/ERAD/view/ERAD NE 2011 ForumPG.pdf.

UM AMBIENTE COMPUTACIONAL TOLERANTE A FALHAS PARA APLICAÇÕES PARALELAS

Oberdan Rocha Pinheiro

Salvador, Fevereiro/2013.